DISKETTE INCLUDED

ADVANCED INFORMIX-4GL PROGRAMMING

INFORMIX® Press

ART TAYLOR

INFORMIX® Press

ADVANCED INFORMIX-4GL PROGRAMMING

Art Taylor

PRENTICE HALL P T R ENGLEWOOD CLIFFS, NJ 07632

Library of Congress Cataloging-in-Publication Data

Taylor, Art
Advanced INFORMIX-4GL programming / Art Taylor
p. cm.
Includes index.
ISBN 0-13-301318-9
1. INFORMIX-4GL (Computer program language) I. Title
QA76.73.I22T39 1995 94-22197
05.75'65 dc20 CIP

© 1995 Prentice Hall P T R

Prentice Hall PTR
Prentice-Hall, Inc.
A Simon & Schuster Co.
Englewood Cliffs, New Jersey 07632

INFORMIX Press

Informix Press
Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

Editorial production/supervision: *Mary P. Rottino*
Cover design director: *Jerry Votta*
Cover design: *Jeannette Jacobs*
Acquisitions editor: *Mark L. Taub*
Editorial assistant: *Marcy Levine*

Manager, Informix Press: *Todd Katz*
Founder, Informix Press: *Suzanne Fuery*

The publisher offers discounts on this book when ordered in bulk quantities. For more information: Corporate Sales Department, Prentice Hall PTR, 113 Sylvan Avenue, Englewood Cliffs, NJ 07632. Phone: 201-592-2863; FAX: 201- 592-2249.

INFORMIX-4GL, INFORMIX-DBA, INFORMIX-ESQLC, INFORMIX-Hyper-Script, INFORMIX-Hyper-Script Tools, INFORMIX-NET, INFORMIX-OnLine, INFORMIX-STAR, INFORMIX-SQL, INFORMIX-View Point, INFORMIX-Wings are registered trademarks of Informix Software, Inc. InterNet, Microsoft Windows, Microsoft Windows NT, Oracle, Prodigy, UNIX, and USENET are registered trademarks of their respective owners.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN 0-13-301318-9

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Dedication

To Carolyn, Hannah and Eric

Contents

Preface ..... **ix**
About this Book ..... **xi**
Modular Programming with INFORMIX-4GL xi
Book Structure xii
Intended Audience xii
Future 4GL Upgrades xii
Not Covered in This Text xii
Book Summary xiii
Conventions Used in This Guide xv
Chapter 1: Introduction to INFORMIX-4GL **1**
The INFORMIX-4GL Language 1
Statements and Operators 2
Program Blocks 2
Language Operators 2
Memory Control 3
Data Typing 3
Database Interaction 3
Parts of a 4GL Application 3
Modular Programming with INFORMIX-4GL 6
Chapter 2: Data Types and Operators **7**
Data Types 7
Conversion Operations in INFORMIX-4GL Assignments 16
Assignment and Conversion in Expressions 18
Evaluation of Looping Statements in INFORMIX-4GL 21
Evaluation of Boolean Expressions in INFORMIX-4GL 22
Chapter 3: SQL and INFORMIX-4GL **25**
Type Conversion with SQL Statements 26
Database Binding in INFORMIX-4GL 27
prepare Statement 29
Optimization with the prepare Statement 31
SQL Processing: Front-End vs Back-End Activity 31
Optimization with the prepare and declare Statements 32
Cursors in INFORMIX-4GL 35
The foreach Statement 35
INFORMIX-4GL Data Variables Used in SQL Statements 35

Cursors in INFORMIX-4GL 37
Types of Cursors 37
Scroll Cursor 38
Cursor Identifiers 45
Closing the Cursor 45
Views 45
Modularity and SQL 47
Chapter 4: Screen Forms and INFORMIX-4GL 49
Screen Forms 49
Managing Screen Size 49
Screen Form Programming 49
Database Binding in Screen Forms 50
formonly Screens 50
Manipulating the Screen Form in INFORMIX-4GL 52
Modularity and Screen Form Interaction 52
The Current Window 53
display Statement 58
input Statement 62
Termcap Issues 66
Trapping the Interrupt Key 68
Chapter 5: INFORMIX-4GL Reports 71
Report Statements 72
First Page Header 77
format Section—before group of/after group of Clauses 79
Batch Processing in Reports 80
print Statement 82
Managing Text Fields in Reports 83
Page Breaks 84
Modular Dynamic Reports 86
Report Issues 88
Modularity in Reports 90
Synchronous Reports 92
Dynamic Reports Driven by the Construct Statement 93
The acct_totals Report 96
Table-Driven Dynamic Report 100
The on every row Report Block 108
Chapter 6: Functions in INFORMIX-4GL 111
Implementation 112
Use of Variables Within a Function 114
Passing a Variable Number of Parameters to Functions 116

Recursion in INFORMIX-4GL....119
Towers of Hanoi Problem....120
An Example of Recursion — The Parts Explosion Problem....132
Generic Functions Using the Variable Parameters Example....137
Passing a Variable Number of Function Parameters in INFORMIX-4GL...140
Adding Additional Functionality with Generic Functions....145
Summary....147
Chapter 7: Input Array and Display Array: Pick Lists and Zoom Functions....149
Display Array Examples: The Zoom Functions....149
input array Examples:
Generic input array Functions with On-Screen Sorting....158
Program main and globals Section....172
Searchable Display Array....181
The searchit Function....183
Chapter 8: Additional Screen Examples: Calculator and Calendar....191
The Calculator....191
The Calendar Application....209
Chapter 9: Table Browser Application....217
The Program Main....218
The globals Section....221
The gen_tab_input Function....222
Menu Options....224
Moving Forward a Page....254
Moving Back a Page....255
Summary....264
Chapter 10: Error Handling and Help Messages in INFORMIX-4GL....265
whenever error Statement....266
Error Logging with INFORMIX-4GL....268
Trapping Forms and Expression Errors with whenever any error....270
Error Messages....271
Generalized Error-Handling Routines....272
An Error Prompt Function....273
Help Messages in INFORMIX-4GL....274
The sqlca Record....276
The sqlerrd Array....277
The sqlwarn String....278
Summary....279

Chapter 11: An INFORMIX-4GL Template Application 281
The main Program Block 282
Converting the Template Application to Process Other Tables 294
Summary 295
Chapter 12: Optimizing INFORMIX-4GL Applications 297
Database-Bound Applications 297
Optimizing INFORMIX-4GL Loop Operations 301
Optimizing Function Calls 302
Eliminating Unnecessary Statements and Expressions 303
Reducing String Operations 304
Eliminating Parameters 307
SQL Optimization 308
Summary 308
Chapter 13: An INFORMIX-4GL Development Environment 311
Libraries 311
UNIX Makefiles 312
Supporting P-code INFORMIX-4GL and Compiled C-code Versions with Make 314
Source Code Control Support in Makefiles 315
Recognizing Directories in a Makefile 316
Summary 316
Chapter 14: General Function Library Routines 317
Bubble Sort Routine 326
Chapter 15: INFORMIX-4GL C-Language Wrapper Functions 335
The INFORMIX-4GL Stack 335
INFORMIX-4GL C Library Access Functions 338
Summary 351
Index 352

Preface

As this book goes to print, an important set of improvements is being applied to INFORMIX-4GL. These improvements are designed to make the language more dynamic and flexible. Several important features, such as the raising or elimination of limits, call-by-reference, dynamic memory management and object-oriented capabilities, are being added. Coupled with the object-oriented features and built-in object libraries are flexible, powerful GUI capabilities. These features make what has always been a powerful business language even more so. They are destined to once again carry INFORMIX-4GL to the forefront of business programming languages.

But within the object methods and GUI extensions, the core language which has been presented in this text is retained. All of the material presented in this text is still valid, and mastering this portion of the language will continue to be important.

All the modular or object-oriented design methods have limited impact if the programmer does not understand and use the core capabilities of a language to their fullest. As with any programming language, it is up to the programmer to use it correctly; the language cannot, for the most part, make that happen. To that end, a knowledgeable, experienced programmer will make best use of a language. It is hoped that this book will contribute to this important skill development effort for the INFORMIX-4GL language.

I would like to thank those who helped with this book. Specific thanks to Bill Hedge for providing useful information on the sources and future of the INFORMIX-4GL language; to Gopi Reddy for clarifying several important technical issues; and to Jim McCann and Andrew Lanthier, whose perceptive technical review have undoubtedly made this a better text.

—Art Taylor

About this Book

Since its introduction in the mid-1980s, INFORMIX-4GL has come to dominate the UNIX® 4GL market. Originally designed as a COBOL replacement, its speed and flexibility have allowed it to be used on numerous large-scale mainframe replacement applications where previously only COBOL or some other 3GL would have been seriously considered. This popularity is not accidental. In short, INFORMIX-4GL provides the flexibility of a 3GL with the power and ease of use of a 4GL.

The goal of this book is to demonstrate the power of INFORMIX-4GL by approaching the language as an advanced programming tool. As part of this approach, the book provides complex examples of INFORMIX-4GL code for a variety of general-purpose applications such as a calendar, a table browser and a desktop calculator. Also shown will be advanced reporting techniques to produce dynamic reports.

Because of its advanced features, such as recursion, complex data types and multiple function parameters, INFORMIX-4GL has the capability to provide solutions to problems previously reserved only for 3GLs. Solutions to problems such as "The Towers of Hanoi" and "The Parts Explosion Problem" are provided in this book. Most code examples are working programs which can be compiled and run. A set of toolkit library functions are also provided to create pick lists, a calculator and a calendar.

Modular Programming with INFORMIX-4GL

The benefits of modular programming are in step with those of all 4GLs: application development time is reduced, code maintenance is made easier and programmer training time is shortened. Much of the benefit of using INFORMIX-4GL derives from its modular, block-oriented language structure. The language was designed to be used in a structured development environment to develop structured applications. Many programmers familiar with the COBOL style of programming tend to use INFORMIX-4GL in an unstructured manner. This could lead to redundant, repetitive code which provides little or no reduction in code over a 3GL. Instead of the 10:1 reduction in application code often claimed of 4GLs, these programs provide very little improvement over using an unstructured 3GL such as COBOL.

Though there is no requirement to use INFORMIX-4GL to create modular programs, there are obvious benefits to doing so. Failure to create INFORMIX-4GL programs that adhere to the principles of modular application development can significantly reduce the benefits of using the language. Throughout this book, the principles of structured application development will be stressed. Only INFORMIX-4GL syntax and programming styles that are amenable to this approach will be shown.

Book Structure

The first few chapters of this book cover the basic statements and operators of INFORMIX-4GL. The remaining chapters cover the more advanced concepts of INFORMIX-4GL programming. The object of this structure is to provide a thorough explanation of an INFORMIX-4GL statement or operator and provide tips for its usage before it is used in an example.

Intended Audience

This book assumes a basic product knowledge of INFORMIX-4GL. If the user does not have this knowledge, a review of the INFORMIX-4GL tutorial provided with the product is recommended.

This book is not intended to replace INFORMIX-4GL documentation; it is meant to complement it and provide additional information for the user who intends to push the product to its limit. The Informix product documentation is highly recommended to any INFORMIX-4GL user.

Because of the completeness of INFORMIX-4GL documentation, there is bound to be some overlap of the material in this book with the material covered in the INFORMIX-4GL documentation. This book will devote more extensive coverage of some areas that are lightly covered in INFORMIX-4GL documentation and will offer additional examples for many of these topics.

Future 4GL Upgrades

The examples in this book were developed and tested with INFORMIX-4GL version 4.0 under various flavors of UNIX. As this book goes to press, an upgrade to INFORMIX-4GL is being developed. The functions and features of this upgrade have been considered in the preparation of this text. The operations, code examples and programming style shown in this book are still valid with these features. Where new features or functionality have a bearing on the discussion, they will be identified.

Not Covered in This Text

The focus of this book is on syntax and coding complex algorithms with INFORMIX-4GL. Since there is little or no difference in syntax, this book does not explicitly discuss the minor variations of INFORMIX-4GL RDS, RF, GX or Asian Language products. In any discussion of the language provided in this text, it can be assumed that all versions are being discussed.

Additionally, the INFORMIX-4GL programmers environment is not discussed. Once again, since the focus is language use, use of this development environment tool does not have much bearing on this discussion.

BOOK SUMMARY

Chapter 1 — Introduction to INFORMIX-4GL

This chapter will provide an introduction to the basics of INFORMIX-4GL. Procedural statements are explained and contrasted with INFORMIX-4GL non-procedural statements. The steps involved in developing an INFORMIX-4GL application from the ground up are identified.

Chapter 2 — Data Types and Operators

This chapter begins with a discussion of INFORMIX-4GL data types. Data types are listed, along with minimum and maximum values and tips on usage. Data conversion rules are also discussed. INFORMIX-4GL operators and order of evaluation are then identified and described along with order of execution.

Chapter 3 — SQL and INFORMIX-4GL

This chapter will discuss using SQL statements in an INFORMIX-4GL program. How SQL is integrated into the language and current limitations are discussed. The usage of cursors in INFORMIX-4GL, dynamic SQL and current language limitations in this area will be discussed, as well as the INFORMIX-4GL `foreach` statement.

Chapter 4 — Screen Forms and INFORMIX-4GL

Forms are the backbone of screen input/output operations in INFORMIX-4GL. How 4GL statements interact with screen forms will be discussed. This chapter will discuss how screen forms are designed, how to perform special operations with screen forms, and how 4GL statements interact with screen forms. Some current limitations will be identified and workarounds to these limitations will be demonstrated.

Chapter 5 — INFORMIX-4GL Reports

This chapter will list basic INFORMIX-4GL report statements. Report-sorting versus SQL-sorting will be discussed. Using the INFORMIX-4GL report writer for batch processing, an example of report driven by a `construct` statement as well as a dynamic, table-driven report are also presented.

Chapter 6 — Functions in INFORMIX-4GL

Functions are an integral part of modular programming with INFORMIX-4GL. How functions are executed within INFORMIX-4GL, what conversion rules are applied and how to take advantage of these rules to develop generic functions will be discussed.

This chapter will also show how recursion is implemented using INFORMIX-4GL and how it can be used to solve real-world problems. A solution to the "Parts Explosion Problem" is provided, as well as an INFORMIX-4GL solution to the "Towers of Hanoi" problem. A scheme to handle a variable number of function parameters in a generic function is also demonstrated.

Chapter 7 — Input Array and Display Array: Pick Lists and Zoom Functions

This chapter discusses a very important part of INFORMIX-4GL, screen input/output, in detail. Examples of using `input array` and `display array` statements to develop pop-up windows, pick lists and generic table *zoom* functions are shown. An `input array` list with on-screen sorting, a sideways scrolling array and a searchable display array are also presented.

Chapter 8 — Additional Screen Examples: Calculator and Calendar

This chapter continues the review of screen interaction in INFORMIX-4GL by presenting two more complex examples. A calculator application and a calendar application are shown and discussed in detail.

Chapter 9 —Table Browser Application

This chapter presents a table-browsing application, a complex application that combines dynamic SQL with dynamic screen form usage.

This application is passed the name of a database and allows the user to choose a table name from a pick list of tables available in the database. Once the table is chosen an input screen is displayed, and the user is allowed to add, update, modify, delete or browse a list of records from the table chosen. The application demonstrates some of the dynamic capabilities of INFORMIX-4GL; the SQL queries and much of the screen form interaction is dynamic — built entirely at runtime.

Chapter 10 — Error Handling and Help Messages in INFORMIX-4GL

This chapter discusses the error handling and help message facilities of INFORMIX-4GL. Several examples of error handling schemes and display of help messages are presented.

Chapter 11 — An INFORMIX-4GL Template Application

Using template applications is a common approach to programming with INFORMIX-4GL. This chapter presents a template application to browse and update a table or set of tables.

Chapter 12 — Optimizing INFORMIX-4GL Applications

This chapter provides tips and examples for optimizing INFORMIX-4GL applications. Several examples are presented to demonstrate the concepts presented in the chapter.

Chapter 13 — An INFORMIX-4GL Development Environment

This chapter shows a common INFORMIX-4GL development environment. Several examples of using the `make` utility with INFORMIX-4GL are shown.

Chapter 14 — General Function Library Routines

A good toolkit of functions is useful with any language; INFORMIX-4GL is no exception. This chapter presents a number of useful functions for string manipulation, searching strings, and displaying prompts and error messages. A working example of a bubble sort routine and a soundex coding routine are also presented.

Chapter 15 — INFORMIX-4GL C-Language Wrapper Functions

The C language is powerful not only for its syntax and flexibility, but for its wealth of library functions. This chapter demonstrates the ease with which these library functions can be accessed using the INFORMIX-4GL facility for interfacing with C functions. INFORMIX-4GL functions which provide access to the C-language library functions for `puts`, `gets`, `open`, `close`, `getpid`, `unlink` and `rand` are shown.

Conventions Used in This Guide

This book uses the following notational conventions.

- Code examples, program elements and examples of literal program input or output are shown in fixed-width font (Courier). e.g., `This is how code text appears when used in this book.`
- Occasionally, code example lines will not fit on one line. When they do not an oversize "break" symbol is inserted (⇐). The line is then broken, moved to the following line and indented. For example:

```
update accounts
   set ( acct_group,acct_type,acct_name)  =
       ( appl_rec.acct_group, appl_rec.acct_type, appl_rec.acct_address⇐
      appl_rec.acct_name )
```

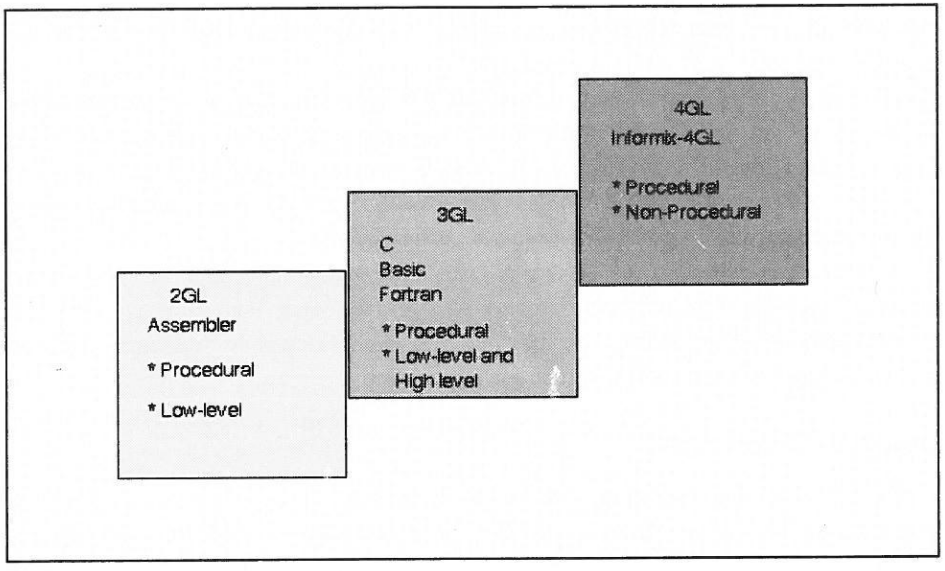
Chapter 1

Introduction to INFORMIX-4GL

THE INFORMIX-4GL LANGUAGE

4GLs were designed to reduce the effort needed to program an application. Figure 1-1 shows the history of the development of language generations. They were intended to make this effort easier through the use of *nonprocedural* statements. Whereas *procedural* statements require the programmer to tell the computer specifically how to do something, *nonprocedural* statements can be used to simply tell the computer what to do; it is up to the language to handle the details. Examples of procedural statements are flow-of-control statements such as `while` and `for`, and decision operators such as `if` or the `case` statement. Examples of nonprocedural statements in 4GL are the `display form` statement.

Figure 1-1 The History of Language Generations



A key benefit of INFORMIX-4GL is its optimal mix of procedural and nonprocedural statements. As the designers of INFORMIX-4GL knew, a 4GL cannot depend on nonprocedural statements alone. In order to be flexible and provide sufficient functionality to develop a complex application, a language must contain an adequate

number of procedural statements. This is where many 4GL products have come up short. Without an adequate set of procedural statements available, a 4GL programmer would be forced to resort to 3GL statements in order to complete the application; this is a tedious process that quickly eliminates any advantage of using a 4GL in the first place.

The INFORMIX-4GL language is composed of *statements*, *blocks*, *operators* and *functions*. An INFORMIX-4GL program is any arbitrary combination of any of these with a valid entry point and a valid exit. The valid entry point is through the main program block, and the valid exit is through either the end main statement that indicates the end of the main program block or the exit program statement.

STATEMENTS AND OPERATORS

INFORMIX-4GL statements control program operation. They can be flow-of-control, decision operation or nonprocedural statements. A block can be a *program* block, a *function* block or a *report* block.

PROGRAM BLOCKS

Of the program blocks, only the main block is required; all other INFORMIX-4GL blocks are optional. There are no requirements for other program blocks, though adhering to the principles of modular programming would certainly encourage the use of the function block.

At runtime, blocks are executed in a serial fashion with the main program block being the first to be executed. Additional blocks, such as a function block or a report block, are executed when the program flow of control passes control to one of these blocks. Program execution finishes when an exit program statement is encountered, or when the end main-block-terminating statement is encountered.

A *function* is composed of all the language statements and operators between the function declaration (`function function_name`) and the `end function` statement. More than one function can be contained in a source code module. A function declaration cannot be made within a function declaration.

LANGUAGE OPERATORS

INFORMIX-4GL operators can be mathematical operators such as addition, subtraction, multiplication, division, module or string-oriented operators such as the string concatenation operator. Operators can optionally be grouped explicitly using parenthesis. Order of precedence and a description of operators are provided in Chapter 2.

MEMORY CONTROL

INFORMIX-4GL controls memory carefully, providing a level of safety not generally available in C. In this way, INFORMIX-4GL's design stays closer to PASCAL

than C. The programmer does not have free and easy access to memory. While this may seem a hindrance to some used to the free-wheeling access to memory provided by the C language, it does eliminate the need to spend several days tracking down an infamous C language *memory leak*.

DATA TYPING

INFORMIX-4GL provides stricter data typing than most 4GLs. In INFORMIX-4GL, variables must be declared to be a specific data type before they are used. Unlike many 4GLs, variables do not simply spring into existence, with data type implied by assignment statements. This provides a level of control for the programmer. The programmer can be reasonably confident of the value and characteristics of a variable at a particular point in time.

Strict data typing can be constraining when data conversion is needed. That is not the case in INFORMIX-4GL; data conversion is implied during assignment statements. In order to convert the data type of a value contained in a variable, the programmer need only make an assignment of the variable to a variable of the desired data type. This is an extremely useful language feature, which will be capitalized upon throughout this text.

DATABASE INTERACTION

The INFORMIX-4GL language is tightly bound to the Informix database. INFORMIX-4GL interacts with the Informix database using the industry-standard Structured Query Language (SQL).

All SQL statements interact only with the Informix database. Currently, no other databases are supported. The SQL syntax is virtually identical to the SQL used in other Informix products, with additions being used primarily to support the use of INFORMIX-4GL variables as SQL host variables.

PARTS OF A 4GL APPLICATION

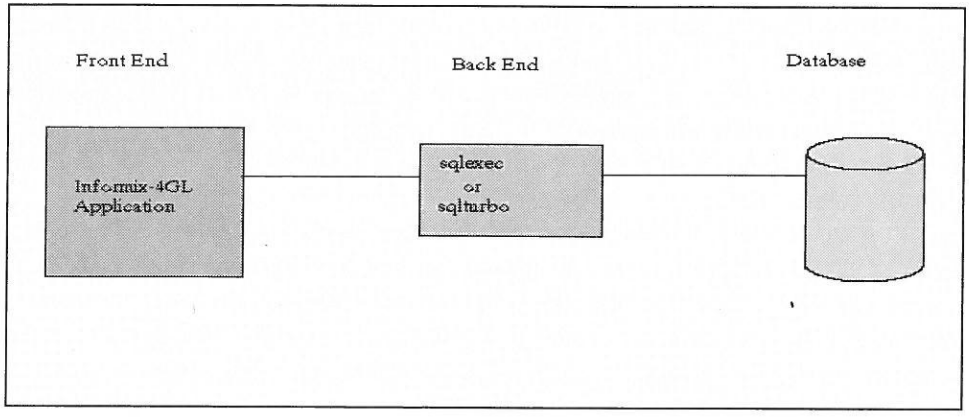
INFORMIX-4GL Runtime Application

INFORMIX-4GL language statements can have an impact on both the *front end* and *back end* of an application. The front end of an INFORMIX-4GL application is the application program itself. This contains the executable portion of the application and the data variables that have been assigned within the program. Most INFORMIX-4GL procedural statements and a number of nonprocedural statements affect only this portion of an INFORMIX-4GL application.

The INFORMIX-4GL back end is the portion of the program that processes SQL statements. In all versions of the Informix database engine up to version 6.0, an additional

process was spawned if a database operation was executed. This process is named `sqlexec` with the Standard Engine, or `sqlturbo` for the On-line engine (see Figure 1-2). The purpose of this process is to communicate with the database engine when On-line is used, or to retrieve the data if the Standard Engine is used.

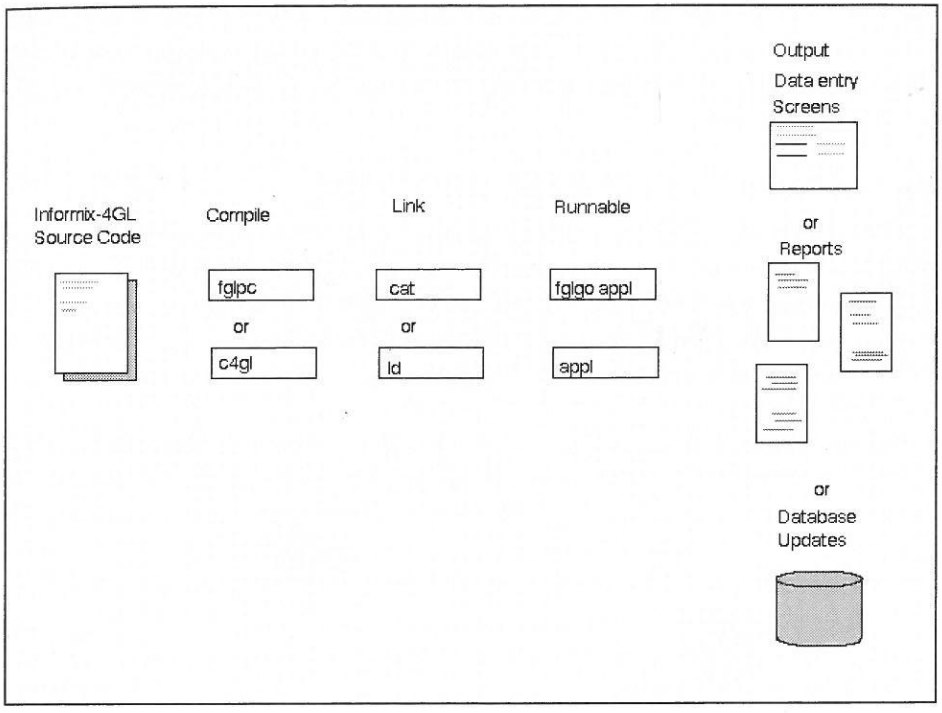
Figure 1-2 The INFORMIX-4GL Runtime Application



The Program ModuleAn INFORMIX-4GL application is composed of a set of source code *modules*. A source code module contains a set of statements, data variables and operations performed on the data variables. Only the main block statement is required in at least one of the source code modules; all other statements are optional. The modules are compiled and linked together to form the executable INFORMIX-4GL application. This process can be streamlined using the UNIX `make` utility or the INFORMIX-4GL programmer's environment.

There is no specific rule for the composition of INFORMIX-4GL program modules. A particular module is usually composed of a set of function blocks. Each module then contains a group of INFORMIX-4GL functions, which should all relate to some activity being performed by the application (see Figure 1-3).

Figure 1-3 INFORMIX-4GL Development



Adhering to the rules of modular programming would produce a program that is composed of a set of source code modules, each module containing a set of statement blocks. There would, by definition, be only one main statement block. There should be multiple function blocks spread across several different program modules. Each function block would have a limited number of program statements and operations. The size of each function would be kept small to improve readability and maintainability of the application.

The Screen Form

INFORMIX-4GL performs the majority of its screen interaction using the screen form. This screen form is virtually identical to the screen form used with the Informix-Perform/Informix-ISQL product. This obviously allowed the developers of INFORMIX-4GL to reuse code originally used for the Perform product and had the added benefit of providing an easy migration path for Perform users, who then had the option of upgrading to INFORMIX-4GL using their existing screen forms.

Screen forms also add controls to the data entry process. Data types are automatically validated and are usually directly bound to the database tables for data type and size. However, there are times when a generic data entry form is needed, one that is not bound to any database tables. Methods of making use of such generic forms are demonstrated in this text.

MODULAR PROGRAMMING WITH INFORMIX-4GL

INFORMIX-4GL provides a balance of procedural and nonprocedural statements; and it does so with block-oriented statements that are designed to promote a modular method of application development. Though meant as a COBOL replacement, it does not contain any of the modular programming failings of COBOL. In fact, its design borrows more from such block-oriented modular languages as PASCAL, ALGOL and C than COBOL or FORTRAN.

The benefits of modular programming have been known and documented for some time. INFORMIX-4GL was written with modular programming in mind. The language is block-oriented and contains a variety of both structured procedural and structured nonprocedural statements that make modular application development easier.

Modular programming requires that an application be developed from a set of small to medium-sized source code modules. Complexity is managed by keeping the size of modules small and encapsulating functionality in small, reusable functions. Many of these functions could be considered black-box functions; they contain code with which the programmer need not be concerned. The programmer need only be concerned with how to use the functions, not with the specifics of their operation. The programmer only needs to know what parameters will be passed into the function and what parameters will be returned.

INFORMIX-4GL was designed to be used in this manner. Its full-featured implementation of functions provides the capability to receive multiple function parameters of virtually any valid data type and return any number of parameters. This makes the development of modular function libraries easy. This concept of modular programming is a cornerstone of this text.

Chapter 2

Data Types and Operators

DATA TYPES

INFORMIX-4GL supports the full range of data types supported in 3GL languages. With a few exceptions, INFORMIX-4GL supports all of the data types supported by the C programming language. In fact, INFORMIX-4GL adds a number of additional data types.

INFORMIX-4GL simple data types have a one-to-one correspondence with the data types supported by the Informix database, including the Binary Large Object (Blob) data type. (More complex data types such as arrays and records obviously do not have this correspondence.) Table 2-1 lists all INFORMIX-4GL data types along with their storage size and their minimum and maximum values.

Table 2-1 Valid INFORMIX-4GL Data Types

Data Type	Length	Minimum Value	Maximum Value
smallint	2	-32,767	+32,767
integer, int	4	-2,147,483,647	+2,147,483,647
smallfloat, real	4	platform-dependent	platform-dependent
float, double precision	8	platform-dependent	platform-dependent
decimal,dec, numeric	precision/2 + 1	-32 significant digits	+32 significant digits
money	precision/2 + 1	-32 significant digits	+32 significant digits
date	4	1/1/0001	12/31/9999
char, character	programmer-defined	1 byte	32,767 bytes

Smallint

The INFORMIX-4GL `smallint` is a two-byte integer and corresponds to the C short integer. It is signed and supports a minimum/maximum value of +/- 32,767. An integer overflow is an error in INFORMIX-4GL. In the p-code/RDS version of

INFORMIX-4GL, this error is trapped by default. In C-code-compiled INFORMIX-4GL, this error is only trapped if the `whenever any error` statement is in effect.

Integer

The INFORMIX-4GL integer is a four-byte integer that corresponds to the C `long` integer. It is also signed and supports a minimum/maximum value of +/- 2,147,483,647. Integer overflow errors are trapped as they are with the smallint data type.

Unfortunately, there are no macro constants provided to test for maximum integer values; the programmer must store these values in variables and check them accordingly. If a screen form is being used to get these values, the form will automatically validate the numeric data being entered. It will return an error code if a number greater than the maximum allowed value is entered.

Smallfloat

The INFORMIX-4GL smallfloat is roughly equivalent to the C float or real. It has a minimum and maximum value that is officially machine-dependent but is generally on the order of +/- 10^{16}.

Decimal

Because of the difficulties and variations associated with using floating-point numbers on early UNIX machines, Informix developed their own form of floating-point number representation: the `decimal` data type. This design move had the added advantage of making it easier to move data over the network, since the numeric data was stored in a consistent format from machine to machine.

Internally in the INFORMIX-4GL program, the decimal data type is stored as shown in Table 2-2:

Table 2-2 Decimal Data Type Storage

Component	Storage
Exponent	2-byte integer
Decimal position	2-byte integer
Number of digits	2-byte integer
Digits	Character with a length of declared precision/2

The decimal data type can be either fixed-point in representation if a precision is specified during the declaration process, or floating-point if no precision is specified. Its precision is very large, on the order of +/- 10^{32} or 32 significant digits. The INFORMIX-4GL declaration for this data type is

```
define dec_var     decimal
```

Since this declaration does not specify precision, this variable will have the default of 16 digits of precision and the decimal point will float. If, however, the variable had been declared as

```
define dec_var decimal(10)
```

it would create a decimal variable with a maximum number of 10 significant digits; or the variable could have been defined as

```
define dec_var decimal( 10, 2 )
```

This would create a decimal variable of ten digits with a precision of two (two digits to the right of the decimal place).

If m is the precision and n the scale, the largest value that can be placed in a decimal data variable declared

```
define dec_var decimal(m,n)
```

would be $10^{m-n} - 10^{-n}$. This means that a decimal could be declared

```
define dec_var decimal(2,0)
```

This variable could then store a maximum of $10^{2-0} - 10^{-0}$, or 99(100-1). Any attempt to store a number larger than this would generate an expression error, which should be trapped if the whenever any error mode of error checking is in effect (current versions of INFORMIX-4GL may not trap this error by default).

Money

The `money` data type is a special case of the decimal data type. It is, by default, a fixed decimal number with a scale of two. When the money data type is displayed, the default currency symbol is displayed to the left of the digits being displayed. The default output currency symbol may be changed by using the DBMONEY environment variable, which has the format below.

```
DBMONEY=scb
```

In the above format, `s` is the symbol that will precede the money value, `c` is the symbol that will separate the whole currency portion from the fractional portion, and `b` is

the symbol that will follow the money value. The valid values for `c` are "." or ",". The valid values for `s` or `b` are any series of characters that do not include a comma or a period.

Float

The float data type in INFORMIX-4GL usually has the same precision as a C double on the machine platform. Once again, the precision is machine-dependent but is usually limited to 10^{32}. The language-definition `double precision` is a synonym for `float`.

Date

The INFORMIX-4GL date is represented internally as a four-byte integer which represents the number of days since January 1, 1900, inclusive. INFORMIX-4GL automatically formats this value according to either its default format or the format specified by the DBDATE environment variable. The DBDATE environment variable is set as

```
DBDATE=dmync
```

where `d` is the day being set, `m` the month being set, `y` the year, `n`, the number following the year position, the number of digits of the year to print it can be a two or a four; and `c` is the character used to separate the components of the date.

Datetime

The datetime data type is a means of representing an instant in time; it contains both date and time components. The datetime data type has a variable degree of precision. The user can define a datetime variable to be of a particular precision, but then dynamically change the precision as needed. The datetime data type is composed of the fields year, month, day, hour, minute, second and fraction of a second. This data type is very handy for applications that require a timestamp to be stored with a record. The datetime data type is declared as follows:

```
define variable_name datetime first to last
```

In the above example, the identifier *first* or *last* is one of the fields from Table 2-3, where the level for a *last* entry cannot precede a *first* entry from the list in the same datetime definition.

Table 2-3 Datetime Identifiers

Field	Valid Entries	Level
Year	Number from 1 to 9999	1
Month	Number from 1 to 12	2
Day	Number from 1 to 31, depending on month	3
Hour	Number from 0 to 23	4
Minute	Number from 0 to 59	5
Second	Number from 0 to 59	6
Fraction	5 decimal digits	7

Interval

The `interval` data type is used to represent a span of time. It represents the difference between two datetimes. It also supports a varying degree of resolution, which can be defined when the variable is defined or changed at runtime.

Because the `interval` data type is specifically designed to represent a span of time independent of dates, it must be represented by specific components that cannot mix dates and times. There are two interval types: those that represent a year/month span of time, and those that represent day/hours/minutes/seconds. An interval would be declared with the following format:

```
define interval_var interval first to last
```

In the above example, the identifier *first* or *last* is one of the fields from Tables 2-4 and 2-5, where the level for a *last* entry cannot precede a *first* entry from the list in the same datetime definition.

Table 2-4 Year/Month Intervals

Field	Valid Entries	Level
Year	Number 1 to 9999 (default) Number 1 to 999,999,999 (max)	1
Month	Number 1 to 99 (default) Number 1 to 999,9999,9999 (max)	2

Table 2-5 Day/Hour Intervals

Field	Valid Entries	Level
Day	Number 1 to 99 (default) Number 1 to 999,999,999 (max)	1
Hour	Number 1 to 99 (default) Number 1 to 999,999,999 (max)	2
Minute	Number 1 to 99 (default) Number 1 to 999,999,999 (max)	3
Second	Number 1 to 99 (default) Number 1 to 999,999,999 (max)	4
Fraction	Number 1 to 999 (default) Number 1 to 99,999 (maximum)	5

The default entries above are system defaults which can be overridden during the declaration of the interval variable. The declaration statement below demonstrates this.

```
define interval_var interval year(5) to month
```

In this example, the interval variable has been defined to allow five digits in the year, instead of the default value of four digits.

Character

The character data type in the INFORMIX-4GL language is a string of characters of up to 32,767 characters in length. It is always blank-padded to its defined length. Though it is usually null-terminated, it does not depend on a null terminator to determine its length. Its length is fixed when the character variable is defined.

Complex Data Types

Unlike many 4GL languages, INFORMIX-4GL supports complex data types. Records, arrays and database binding to variables are available. These complex data types are a big part of the power of the INFORMIX-4GL language.

Arrays

INFORMIX-4GL arrays are homogeneous collections of data elements. Arrays can be declared up to three dimensions. Arrays are fixed in size at compile time and are currently not allowed to grow dynamically.

By default, array bounds are not checked at runtime in compiled C INFORMIX-4GL. However, a special compiler flag can be passed to the compiler which will force the compiler to check array bounds at runtime. Since the checking of array bounds on every array reference operation can become expensive, it is recommended that this flag only be used during the debugging stage and not with the production version of the program.

Whereas arrays of records are allowed, arrays of arrays are not allowed. Arrays cannot be passed as function parameters.

Records

INFORMIX-4GL supports records of non-homogeneous data types of virtually unlimited length. Elements within the record can be any valid simple or complex data type such as records. Arrays may also be record components.

Records may be passed as function parameters. Note that when this occurs, the function will process this using *call by value*; that is, the entire record variable will be pushed and pulled of the function stack.

Database Binding

INFORMIX-4GL allows variables to be bound to database tables at compile time. These variables then have a direct correspondence with the database tables. This increases the safety of the language and virtually eliminates the data type errors that are so common when retrieving data from a database. Database binding takes place when the like keyword is used during variable declaration.

```
define lcustomer like customer.first_name
```

The asterisk character in this declaration is then resolved to all of the columns in the database table, as the example below shows.

```
define lcustomer_rec record like customer.*
```

This statement would produce a record where all of the columns match, in ordinal position and data type, the columns within the database table that has been referenced.

The through clause can be used with the define statement to identify only a subset of a data tables columns to be used in the record declaration. An example of this clause is:

```
define lcustomer record like
  customer.first_name through customer.last_name
```

The benefits of this capability cannot be underemphasized. The ability to declare records in this manner eliminates the need for the programmer to constantly refer to the

database schema for clues on how to declare variables. Whenever possible, this feature should be capitalized.

Note that the database binding that occurs when using this statement occurs during the compile phase. For this reason, if a record is declared like the customer table, the program is compiled, and then the customer table is modified before the program is run, there is a potential for a type mismatch error at runtime. In a structured development environment, database schemas should come under the same, or stricter, controls as source code, thus minimizing the potential for this error.

Null Data Values

The NULL data type is a special case of data values. It represents the *absence* of data and has a different internal value depending on the data type being evaluated. The NULL data type can be used to distinguish between values such as the number zero, blanks, or date zero, and an unknown value for these data types

For instance, a column that stores the balance of an account could have the value zero. This could be an indication that the account had carried a non-zero balance but was set to zero by a transaction. This zero value could be distinguished from a NULL value, which would indicate that the account had not been used.

Using NULLs in INFORMIX-4GL expressions can be problematic. A NULL data value, in any Boolean expression, returns FALSE for the entire expression. For example,

```
define val1, val2 integer
let val2 = 100000
initialize val1 to NULL

if ( val2 > val1 ) then
  display "val2 is the larger number"
else
  display "val1 is the larger number"
end if
```

This Boolean expression would evaluate to FALSE, and the display statement would not display the error message, even though the value of `val2` appears to be larger than the value of `val1`. This is intentional behavior based on the theoretical assertion that a NULL value is the absence of data and cannot have a value greater or less than any other value.

This implies that in order for expressions to behave as a programmer may expect, if it is possible for a data variable to contain a NULL value, then the data variable must be tested *explicitly* for a NULL value within an expression, as shown below.

```
define val1, val2 integer
let val2 = 100000
initialize val1 to NULL


if val1 IS NULL and val2 IS NOT NULL
  then display "val2 is a number and val1 is nothing."
end if
if val2 IS NOT NULL and val1 IS NOT NULL and  val2 > val1 then
  display "val2 is the larger number"
else
  display "val1 is the larger number"
end if
```

A technique sometimes practiced when using numeric values in INFORMIX-4GL programming is to eliminate NULLs in expressions by scanning data values before they are used and mapping NULLs to zero values. This technique is shown below.

```
define val1, val2 integer
let val2 = 100000
initialize val1 to NULL

   ....

if val1 IS NULL then
  let val1 = 0
end if
if val2 IS NULL then
  let val2 = 0
end if


if val2 > val1 then
  display "val2 is the larger number"
else
  display "val1 is the larger number"
end if
```

CONVERSION OPERATIONS IN INFORMIX-4GL ASSIGNMENTS

INFORMIX-4GL performs automatic data conversion in expressions as long as the conversion follows some simple rules. This implicit data conversion allows an integer to be converted into a string with a simple assignment operation. The statement below performs such a conversion.

```
define x    integer
define str  char(10)

let x = 1 / 2 * 3
let str =  x
```

This statement will create a string with the integer value 1.

In some cases, integer values may be right-shifted and prepended with blanks. In many cases, this may not be desirable. This is especially true when the integer string is being stored in the database as a database key and will be sorted when returned. With that being the case, the sorted data will not appear in sensible order. A simple solution to this problem is to format the data so that it is no longer right-shifted when assigned into the string. This can be done with the using clause, a clause generally referred to only in reference to display or printing options. The syntax is

```
let sir = x using "<<<<"
```

The left arrows above indicate that the integer being formatted be right-shifted when placed into the string. The complete set of format operators for the using clause are listed in Table 2-6.

Table 2-6 Formatting Characters with the Using Clause

Format Character	Action
#	Insert blank for zero.
&	Insert 0 for 0.
<	Left shift results.
$	Place $ at end of the number.
*	Insert * for 0.
.	Place period at this position in the stream of numbers.
,	Place comma at this position in the stream of numbers.
( )	Surround negative number in parentheses.

continued on next page

Table 2.6 continued

Format Character	Action
-	Place negative sign in number if the number is negative.
+	Place positive sign in number if the number is positive.
mm	Insert the month number.
dd	Insert the day number.
mmm	Insert the month name.
ddd	Insert the day name.
yy	Insert 2 digit year.
yyyy	Insert 4 digit year.

Using the above formatting characters, in order to place an optional column at a specific position, the format string

```
let x   =  133444555
let str = x using"<<<,<<<,<<<<"
```

would produce the result

```
133,444,555
```

INFORMIX-4GL performs a variety of conversions when it deems necessary. Table 2-7 below details a number of these conversions.

Table 2-7 Assignment Conversion Rules in INFORMIX-4GL

Expression	Default Rule
let string = numeric	Left-shifted, blank-padded.
let integer = string	Acceptable if string is a stream of ASCII characters that represent integers, or else a fatal error may occur.
let integer = decimal	Truncated to integer.

continued on next page

Table 2-7 continued

Expression	Default Rule
let string1 = string2	No errors or warnings if the length of string1 is long enough to hold string2. If the length of string1 is less than the length of string2, truncation occurs but signals a warning, not an error.
let decimal1 = decimal2	Will alter the scale of the left-hand argument as needed as long as precision is not exceeded. If precision is exceeded, a fatal error may occur.
let date = datetime	Datetime must be in the format year to day.

ASSIGNMENT AND CONVERSION IN EXPRESSIONS

INFORMIX-4GL uses a set of internal rules to perform type conversion during expressions. Each expression is parsed, and individual arguments are placed on the INFORMIX-4GL stack along with the data type and length of the argument. As arguments are pulled off the stack, type conversion takes place as long as the conversion is sensible.

Character Conversions

INFORMIX-4GL will convert any valid data type into a character string without generating errors. If truncation occurs, a warning is generated. If these warnings are not trapped, program execution continues. If both sides of the expression are characters, but the left-hand side of the expression is shorter than the right-hand side, then truncation occurs, a warning is generated and program execution continues. The code fragment below demonstrates this concept.

```
define str1   char(5)
define str2   char(10)

let str2 = "0123456789"
let str1 = str2
display "str1 is ", str1
```

This would produce the output

```
01234
```

If the right-hand side of the expression is numeric, it will be placed into the character variable left-shifted. If the string is not long enough to hold the number, then it will be truncated and a warning generated. Program execution will continue. This behavior is true for all numeric data types used on the right-hand side of a character conversion expression.

Numeric Conversion Rules in INFORMIX-4GL

Arithmetic Operators

INFORMIX-4GL supports a full range of mathematical operators. These operators have a specific order of evaluation, which may not be exactly what the user expects. Table 2-8 below lists the valid INFORMIX-4GL operators and their level of evaluation or grouping.

Table 2-8 Grouping of Operators

Operation	Symbol	Evaluation Level
Associative	()	1
Exponentation	**	1
Multiplication	*	2
Division	/	2
Modulus	mod	2
Addition	+	3
Subtraction	-	3

In numeric operations, all operands are converted to the INFORMIX-4GL data type decimal, and the resulting number is decimal. Rules concerning the precision and scale of the result are detailed in Table 2-9. If the result can be placed in the left-hand value, no errors are returned. If significant digits are lost, a fatal error is sometimes returned. Using the `whenever any error` mode of error checking will provide error checking for all expression errors.

The table details the precision and scale of the result of a mathematical operation. Here, p_1 and s_1 are the precision and scale of the first operand, and p_2 and s_2 are the precision and scale of the second operand.

Table 2-9 Precision and Scale of Result of Mathematical Operations

Operation	Precision	Scale
Addition or Subtraction	MIN(32, MAX (p_1 - s_1, p_2 - s_2) + MAX(s_1, s_2) + 1)	MAX(s_1, s_2)
Multiplication	MIN(32, p_1 + p_2)	s_1 + s_2
Division	32	32 - p_1 + s_1 - s_2

Initialization

Note that INFORMIX-4GL variables are, by definition, not initialized when defined. The p-code version of the compiler initializes data space to binary zero. This can cause significant surprise when moving a p-code application to a compiled C version of INFORMIX-4GL. The C language is a language that by definition does not initialize variables, leaving their value indeterminate until assigned; and as most programmers know, indeterminate data values can wreak havoc on an application. Future versions of INFORMIX-4GL will allow initialization during variable declaration.

Lack of Constants

One limitation of INFORMIX-4GL is its lack of constants, both local and global. An obvious workaround is to use a variable in place of a constant. Since the INFORMIX-4GL language will not restrict a variable from modification during compilation, it is up to the programmer to prohibit this behavior. By using a style convention that easily identifies *constant* variables, this task is made easier. The simplest choice is to use only uppercase letters when naming constants (a common C language convention). The example below demonstrates this convention.

```
-- Create global 'constants'
--
globals
  define  MAXRECS integer
  define  MAXARR  integer
end globals

main

  -- Initialize CONSTANTS
let MAXRECS = 100
let MAXARR  = 200
...
```

In the example above, the integer variables MAXRECS and MAXARR are defined as globals. At the start of the main program block, they are initialized to a specific value. From this point on, they can be used much as any other constant would be in the program. Obviously, constants of other data types could be declared, initialized and used in the same manner.

Evaluation of Looping Statements in INFORMIX-4GL

INFORMIX-4GL provides a set of standard looping statements. They provide the common loop operation statements, such as the `while` loop and the `for` loop. They also provide the `foreach` statement, which combines the function of a cursor with that of a while loop. Table 2-10 lists the INFORMIX-4GL looping statements.

Table 2-10 INFORMIX-4GL Looping Statements

Statement	Parameters	Function
for	Initialization expression, limit expression, optional step expression	Perform loop with index variable beginning at initialization expression, looping till value of index variable exceeds limit expression
while	Boolean evaluation expression	Performs loop till Boolean expression evaluates to false
foreach	Cursor name	Performs loop till cursor fetch operation returns not found condition

The `for` Statement

The `for` statement is used to perform a loop for a specified set of iterations. The statement can evaluate complex expressions for initialization and terminating values. The index variable being used may be altered within the loop (though strict adherence to the rules of structured programming would not allow this). The `for` loop can be initialized with the return value of a function call, as in the following example.

```
for n = get_val(a,b,c) to limit_val(a,b,c)
  do_this(a,b,c)
  do_that(a,b,c)
end for
```

In this example, the functions `get_val` and `limit_val` will be called on this first iteration of the loop in order to get the initialization and terminating values for the loop. However, since loop index values can change within the loop, these expressions must be reevaluated on each iteration of the loop. If loop index variables are not being changed within the loop, this is not efficient. Storing the initialization and terminating values in variables will optimize the looping operation. This approach is shown below.

```
let x = get_val(a,b,c)
let z = limit_val(a,b,c)

for n = x to z
  do_this(a,b,c)
  do_that(a,b,c)
end for
```

Evaluation of Boolean Expressions in INFORMIX-4GL

Boolean expressions in INFORMIX-4GL are evaluated as they are with the C programming language. A non-zero value is evaluated as TRUE; a zero value is evaluated as FALSE. For programmers familiar with the C language, this concept should not be difficult to follow. The statement

```
if (1) then
  display "It's true"
end if
```

would display "It's true". But the statement

```
if (0) then
  display "It's false"
end if
```

would produce no output. Because INFORMIX-4GL allows programmers to negate Boolean expressions, the above statement could be written as

```
if NOT (0) then
  display "It's false"
end if
```

Likewise, negating a non-zero value would produce a value of false, as in the expression:

```
if NOT (1) then
```

```
    display "It's true"
  end if
```

This statement would produce no output.

What is interesting about INFORMIX-4GL is its treatment of expressions in Boolean statements. It is a rule that regardless of the Boolean operators in a Boolean statement, all expressions are evaluated. So in the Boolean statement

```
if ( funct1(x,y,z) and funct2(x,y,z) and funct3(x,y,z) )
then
  do_this_funct(x,y,z)
end if
```

all expressions are grouped using the `and` operator. This requires that *all* conditions must test TRUE in order for the statement to be true. With this knowledge, we could then evaluate the statement from left to right and, as soon as a FALSE is evaluated, quit evaluating the statement. We could do this since we know that a single false value will force this statement to fail.

INFORMIX-4GL does not make this assumption. Boolean statements are evaluated from left to right with no conclusions about statement groupings. Unless there are parenthetical groupings, expressions are evaluated two at a time, with the result being applied to the next expression. In order to optimize the above statement, it could be rewritten as follows:

```
if ( funct1(x,y,z) )  then
    if ( funct2(x,y,z) ) then
        if ( funct3(x,y,z) ) then
            do_this_funct(x,y,z)
        end if
    end if
end if
```

In this code fragment, if the expression `funct1` fails, none of the other statements are executed. The same holds true for the expression `funct2`. This provides a certain level of optimization, since the odds are favorable that unneeded statements will not be executed.

The Concatenation Operator

In INFORMIX-4GL, the string concatenation operator is the comma (,). The operator allows two strings to be concatenated using the following syntax:

```
let str1 = str2, str3
```

In INFORMIX-4GL, strings are always padded to their defined length. In order to eliminate this padding, the `clipped` operator must be used. For example:

```
define str1 char(40), str2 char(40), str3 char(40)

let str2 = "012345"
let str3 = "012345"

let str1 = str2, str3
```

In this example, the final statement would be rejected at runtime, since the padded length of `str2` and `str3` combined is greater than the defined length of the target `str1` (40). Using the `clipped` operator, this problem could be eliminated, as shown below.

```
define str1 char(40), str2 char(40), str3 char(40)

let str2 = "012345"
let str3 = "012345"

let str1 = str2 clipped, str3 clipped
```

This statement would execute, since the clipped length is 12, which will easily fit in the 40-byte character string `str1`.

Chapter 3

SQL and INFORMIX-4GL

An important element of the nonprocedural component of INFORMIX-4GL is SQL and its seamless integration into the language. A significant portion of the power of INFORMIX-4GL resides in this integration with SQL. If used correctly, SQL can reduce the amount of procedural code required to program an application and simplify code maintenance. Long, tedious procedural steps can be compressed into a single SQL statement. Complexity can be hidden from the programmer, and business rules can be enforced by the database. With more recent versions of Informix-SQL, additional levels of data integrity can be maintained, and events can be triggered by SQL code. With SQL, the procedural steps necessary to access and maintain the database can be vastly reduced.

Virtually all valid SQL statements are supported in INFORMIX-4GL, with only minor restrictions. These are significant capabilities; but in order to capitalize on these capabilities, it is important that the underlying database be well designed and normalized. If not, the advantages of SQL will be lost, and SQL will be no more useful to the programmer than isam-type function calls.

A small number of Informix-SQL statements are not supported by INFORMIX-4GL. Most of these are administrative in nature, and are recognized and processed by the `isql` and `dbaccess` programs but are not interpreted by the engine. For instance, the `info` statements are not recognized by INFORMIX-4GL. They are as follows:

```
info tables
info status for <table_name>
info indexes for <table_name>
info columns for <table_name>
```

These are administrative statements, but it is not uncommon for INFORMIX-4GL applications to handle some database administrative functions. For these instances, the following `select` statement can be used to retrieve information about a table.

```
select tabname
from   systables
where  tabid > 99
```

Because all Informix system catalog tables have a `tabid <= 99`, this will return a list of the tables in the current database exclusive of the system catalog tables. The

informational statement `info columns for table_name` can be emulated with the following statement.

```
select colname, coltype
from   syscolumns, systables
where  syscolumns.tabid  = systables.tabid and
     systables.tabname = <table_name>
```

If the column allows NULL values, the integer value for the column `coltype` is masked with a specific bit pattern. To eliminate this mask using 4GL, the following statement could be executed.

```
let pcoltype = pcoltype mod 256
```

Otherwise, the statement could make use of the built-in SQL modulo operator, which could be incorporated into the select statement as follows:

```
select colname, mod( coltype, 256 )
from   syscolumns, systables
where  syscolumns.tabid  = systables.tabid and
       systables.tabname = table_name
```

Likewise, the informational statement `info indexes for table_name` can be emulated with the following SQL statement.

```
select idxname
from   systables, sysindexes
where  systables.tabid = sysindexes.tabid   and
       systables.tabname = table_name
```

This statement would return the index name for all indexes for the table name used in the `where` clause.

TYPE CONVERSION WITH SQL STATEMENTS

As with other statements that assign values in INFORMIX-4GL, SQL statements support type conversion when it makes sense. In all cases, any Informix data type can be converted into a character string as long as the character string is long enough to hold the entire value.

The `select` statement retrieves data from the database and assigns values to host variables using the `into` clause. It is not an error to retrieve fewer data elements than there are host variables; INFORMIX-4GL considers this a warning and not an error. This

behavior will be capitalized upon later in this text in using dynamic SQL statements to retrieve data.

During retrieval of data, data conversion occurs freely much as it does during variable assignment. This is demonstrated in the following example.

```
define str1 char(20), str2 char(20), str3 char(20)

-- column int_col contains an integer value
-- column dec_col contains a decimal value
-- column date_col contains a date value

select int_col, dec_col, date_col
into   str1, str2, str3
from   sample_table
```

In this example, all data types are converted according to the defaults identified in Chapter 2. The resulting values are placed into the strings as long as the strings have appropriate length. If the strings are not long enough, asterisks are placed into the string up to the size of the string.

This type of statement and data conversion will be used to create a generic pick list function in Chapter 7.

DATABASE BINDING IN INFORMIX-4GL

One of the most useful features of INFORMIX-4GL is the ability for program variable definitions to be *bound* to database table columns using the `like` keyword. This allows a simple set of statements to ensure data integrity of data being retrieved from the database, as shown below.

```
database mycust

define customer_rec record like customer.*

select customer.*
into   customer_rec.*
where  cust_numb = 1
```

This code allows the customer record for customer number 1 to be retrieved into a program variable without concern of data type incompatibilities. Note that it is a requirement for the database statement to appear before the data binding `define` statement.

The data binding feature is implemented by having the INFORMIX-4GL compiler read the schema for the database table at compile time, determine the data type and length of the variables, and then produce definitions for the variables to match the database table column definitions in ordinal position and type. The compiler **expands** the record definition into its appropriate components. Note that the database statement must precede a `define like` statement in the source code module. The resulting C code generated by the above definition would appear as follows:

Using the schema

```
create table customer (
      first_name char(15),
      middle_i char(1)
      last_name char(15),
      phone_number char(10),
      address1 char(20),
      address2 char(20) )
```

with the INFORMIX-4GL declaration of

```
define customer_rec record like customer.*
```

the generated C code would be

```
struct {
  char first_name[16];
  char middle_i[2];
  char last_name[16];
  char phone_number[11];
  char address1[21];
  char address2[21];
} customer_rec;
```

However, this feature can cause problems when the structure of the data table has changed in data type or order. Since INFORMIX-4GL will quietly convert data types whenever possible, invalid data may be read into the record, and the error may not be readily apparent. Assume that in the following example, the `customer` table was initially defined to have the following schema.

Schema 1

```
create table customer (
        first_name char(15),
        middle_i char(1)
        last_name char(15),
        phone_number char(10),
        address1 char(20),
        address2 char(20) )
```

However, the schema is changed to the following structure.

Schema 2

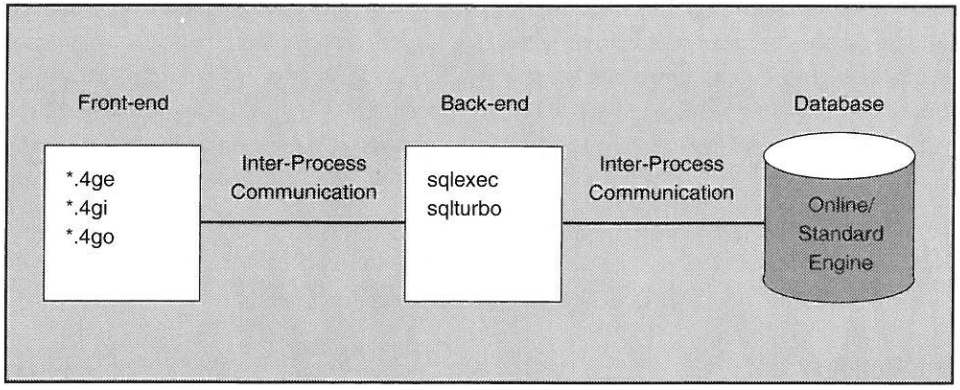
```
create table customer (
        first_name char(15),
        last_name char(15),
        phone_number char(10),
        address1 char(20),
        address2 char(20) )
```

If the program example were **compiled** while schema 1 was in place and then **run** when schema 2 was in place, the data record defined like the schema 1 definition would receive incorrect values. This would not necessarily cause the program to fail. It is likely the program would quietly continue operation, placing the first letter of the user's last name into the `middle_i` component of the record and putting the customer's `last_name` field into the phone number fields. The first `address1` record component would receive the `address2` part of the database record, and the `address2` field in the program record would be untouched; its value would be indeterminate. Since INFORMIX-4GL assumes all data variables are NULL-terminated, attempts to manipulate this component of this record could fail randomly at runtime.

`prepare` STATEMENT

The `prepare` statement serves two purposes in the INFORMIX-4GL language. First, it is used to perform dynamic SQL in INFORMIX-4GL. Using the `prepare` statement, a query can be built dynamically at runtime.

Figure 3-1 Front End to Back End



However, the `prepare` statement is also considered an optimization tool since it can be used to reduce the interprocess communication traffic between the engine and the front end (see Figure 3-1). Since a prepared statement is optimized before it is executed, it reduces the need to continually re-optimize the statement every time it is executed.

To create dynamic SQL, the `prepare` statement can take a string variable as a parameter; within this string variable, virtually any valid SQL statement is supported; the exceptions are the `close`, `declare`, `execute`, `load`, `open`, `prepare` and `unload` statements.

The `prepare` statement merely packages the string it is passed and sends it to the engine for evaluation; it does not examine the SQL statement. All parsing and optimization are performed by the database engine, not in the front end.

There are occasions when the database engine supports syntax not yet supported by INFORMIX-4GL.

The solution to this problem is to place the SQL statement in a string and pass it to the engine using the `prepare` statement, as shown below:

```
let qstr = "execute procedure myproc()"

prepare ex_stmt from qstr

execute ex_stmt
```

In this example, we want to call the stored procedure `myproc`, but the version of INFORMIX-4GL being used does not currently support the `execute procedure` statement. We must therefore place the statement in a string and use the prepare statement to pass the string to the engine. Since the engine recognizes the statement, it will have no problem processing the statement and returning a statement id to the INFORMIX-4GL

program. The INFORMIX-4GL program then issues an `execute` statement for this statement id. Note that if the stored procedure retrieves data, a cursor must be declared for the statement id; otherwise, a runtime error would occur when the statement was executed.

OPTIMIZATION WITH THE `prepare` STATEMENT

When a query is executed in INFORMIX-4GL, the database engine must parse the query, examine the query for validity and syntactical correctness, optimize the query, create temporary tables if needed, access the data files and then begin retrieving data. Not all of these steps may be necessary each time the query is executed.

When the database engine first encounters a query, it must parse the query. Since SQL database objects are not examined until runtime, complete parsing and optimization of the query cannot be done until runtime.

When the database engine checks the query, it will examine the SQL statement for correct syntax (it must do this to determine where the database objects are within the statement). Database objects are examined, and then the database is searched to determne that these objects exist.

In the next step, the engine will optimize the query. Informix version 4.0 engines and later use a cost-based optimizer. This optimizer optimizes queries by examining every possible access path in conjunction with costs for various components of that access path. It chooses the access path with the least cost, and therefore the most efficient access. Since it is quite possible that the elements of the query examined in this step do not change between each execution of the query, it is not necessary to repeat this step. The `prepare` statement can be used to bypass this step.

In transaction-processing environments, optimization with the `prepare` statement is also important because CPU cycles spent preparing a query are CPU cycles that could have been spent processing other applications.

SQL PROCESSING: FRONT-END VS BACK-END ACTIVITY

Informix products were originally designed to make use of a two-process architecture. Though this is changed slightly in version 6.0 of their products, the concept of **front-end** and **back-end** processing is still valid. The **front end** of the application controls the application; the back end performs the database interaction. Certain portions of the SQL statement are processed in the application, and other portions are processed in the engine.

Every SQL statement that is encountered by INFORMIX-4GL is processed in several steps. First, it is parsed to determine the validity of the SQL syntax. Some cursory syntax-checking may be performed by the INFORMIX-4GL compiler in order to ensure

that a valid statement is being used, but the bulk of the SQL statement-parsing will be performed by the database engine.

The parser in the database engine will review the various subclauses of the SQL statement for correctness. The objects that these statements will invariably reference are evaluated based on several criteria. First, it must be determined that they exist in the current database. Then it must be determined that they are valid database objects for the current clause. Next, expressions must be resolved, data type consistencies evaluated, and any necessary conversion performed. Finally, optimization must be performed. The database engine optimizer will evaluate various access paths and an optimal access path will be selected.

OPTIMIZATION WITH THE `prepare` AND `declare` STATEMENTS

If an SQL statement is going to be executed repeatedly, performance can be improved by using the `prepare` statement to parse and optimize the statement before it is used. Alternatively, in the case of `select` statements, the `declare` statement can be used to accomplish the similar optimization.

If the SQL statement is only going to be executed once, then using the `prepare` or `declare` statement does not provide *any* optimization. The following example demonstrates this point. Assume that in this example, the code shown below is executed only once during the application startup procedure.

```
        -- build the query
prepare qstmt from  "select application_name ",
                           "from control_file ",
                           "where control_code = 1 "

declare curs1 for qstmt

        -- fetch the data
fetch curs1 into appl_name
```

In this example, assume the contents of the entire query are known at compile time. There is no benefit to performing a prepare, declare and fetch for a singleton query of this type. It would have been just as efficient to execute the following:

```
-- execute the query
select application_name
into   appl_name
from   control_file
where  control_code = 1
```

The above statement requires fewer lines of code and is clearer; it is easier to read the `select` statement above and understand what is being selected, than to read a lone `fetch` statement, far removed from the code that declared the cursor, and understand what is being retrieved.

Where an SQL statement is being used to validate data being input on a screen, a hardcoded SQL statement versus a prepared SQL statement will probably not produce notable changes. However, in the case of a report that must scan one million rows and execute miscellaneous SQL statements for each row, preparing a statement before repeated execution may make very good sense. The following code samples demonstrate this point. The following example demonstrates the use of the `prepare` statement within the body of a report.

```
on every row

    select fld_stats
    into fld_stats
    from field_stats
    where empl_numb = pempl_numb
```

This query is executed for each tuple passed to the report. If the report is reading and processing a large number of rows, the execution and parsing of this query will slow the execution time of the report. In order to speed up this process, the statement could be prepared as shown below.

```
  first page header

  declare fld_stats_curs cursor for
    select fld_stats
    into fld_stats
    from field_stats
    where empl_numb = pempl_numb

  ....

on every row

    open stmt1
    fetch stmt1 into fld_stats
```

In this example, the statement is prepared in the first-page header block which will only be executed once. Once this block of code has been executed, the optimized statement in the `on every row` block is executed for every row passed to the report.

The steps of unnecessarily evaluating database objects and optimization and the related IPC traffic are avoided in this solution.

In the case of the SQL `update` or `delete` statements, which are usually executed to act on a single row at a time, the `prepare` statement can be used to optimize these operations. For a batch update program that must update numerous rows based on varying criteria, using the `prepare` statement can produce significant performance improvements. This is demonstrated in the following example.

```
on every row

    if ( do_update )  then
    update customer
    set    customer.addr1 = customer_rec.addr1,
    customer.addr2 = customer_rec.addr2

          ....
```

In this example, a report program is being used to perform batch updates based on a flag set elsewhere in the application. Assuming that a large number of rows are being processed, optimizing this update could have a significant impact on this program's execution speed. The following code demonstrates this optimization.

```
first page header

  prepare upd_stmt
     from "update customer set customer.addr1 = ?, ",
           " customer.addr2 = ?  ",
             " where customer_num = ? "
...

on every row

     if ( do_update )  then
  execute upd_stmt using c ustomer_rec.addr1,
                                  customer_rec.addr2,
                                  customer_rec.cust_numb
          ...
```

In this example, the page header block is used to prepare the update statement. When the `on every row` statement is executed, the `do_update` flag is evaluated, and if it is set to TRUE, the update statement is executed via the `upd_stmt` statement id.

In a large batch program processing a large number of rows, this could provide a significant performance improvement.

Optimization should not be limited to using the `prepare` statement alone. Numerous other factors can affect the performance of a query: the existence of appropriate indexes, sort criteria and the duplicity of indexes. If database access is viewed as a bottleneck, these factor should be reviewed.

CURSORS IN INFORMIX-4GL

Cursors are a means of reconciling a row-at-a-time, serial language such as the procedural statements in INFORMIX-4GL, with the set-at-a-time, set-oriented statements of SQL. Cursors provide the means of managing this impedance mismatch of language statements. In the case of a `select` statement, cursors provide a pointer into the active set that is referenced by the select statement.

THE `foreach` STATEMENT

The `foreach` statement combines some of the properties of a procedural statement with those of a nonprocedural SQL statement. A `foreach` statement relates to a cursor and all of the tuples within that cursor; in that respect, it is an extension of a cursor. However, it is also used to create a loop structure similar to that of a `while` loop. It is a `while` loop that loops until a not-found condition is satisfied. At this point, program execution begins at the first statement following the `end` of the `foreach` statement.

A substitute for the `foreach` is a cursor open followed by a `while` loop which continues to loop until a not-found condition is detected. However, the `foreach` statement is a more attractive alternative to programming this logic. The `foreach` statement provides a useful structure for loading arrays scanning large numbers of rows and producing reports.

INFORMIX-4GL DATA VARIABLES USED IN SQL STATEMENTS

INFORMIX-4GL variables used in SQL expressions are evaluated for data type and length at compile time. They are checked for content at runtime. Values to be used in SQL statements can be supplied at runtime using one of several alternatives. The statement can be declared as a cursor using the variables that have been defined as parameters in the where clause of the SQL statement, or the statement can be prepared using the '?' as placeholders for values that will be substituted at runtime.

There is no real performance advantage to using one versus the other. The internal implementation is virtually the same. The only issue to consider is that of programming convenience and style.

One method of supplying runtime values is to simply include the programming variable references in the cursor declaration. This method is shown in the example below.

```
declare cursor c1 cursor for

  select *
  from customer
  where cust_numb = customer_rec.cust_numb

open c1
fetch c1 into customer_rec.*
```

In this example, the cursor is declared with the filter criteria of the SQL `where` clause referencing the INFORMIX-4GL variable record element. This record element will be bound to the cursor. When the cursor is opened, the contents of the host variable are examined and substituted for filter criteria. This method is the *only* supported method of passing parameters into a `foreach` statement, since this statement does not support a `using` clause.

In the next example, the cursor is created with placeholders for the filter criteria.

```
prepare s1 cursor for " select *  ",
                        " from mycust ",
                      " where cust_numb = ? "

declare c1 cursor for stmt1

open c1 using customer_rec.cust_numb
```

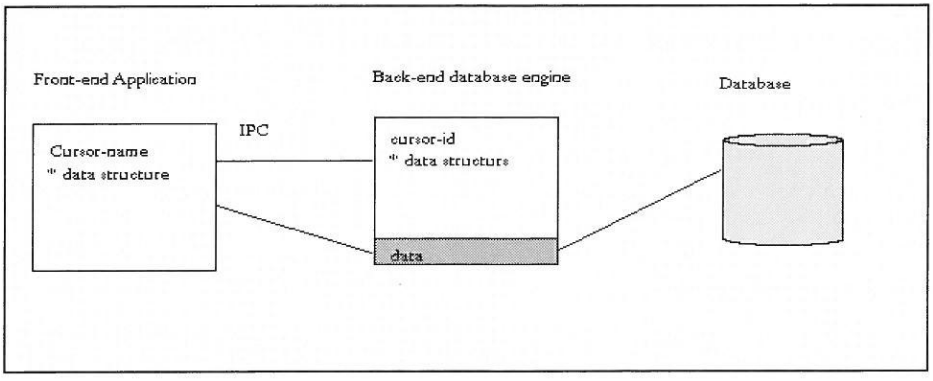
In this example, a statement id is first established for the `select` statement and then a cursor is declared for that statement id. Finally, the cursor is opened with the `using` clause. This clause is necessary in this example since the cursor must access the filter criteria to open the cursor. (Note that in the current version of INFORMIX-4GL, the cursor declaration must precede its use in the source code module.)

There is no single correct way to bind data variables in INFORMIX-4GL; both methods shown are valid and could be considered modular in form. The first example requires slightly less coding but the second example makes it clear in the `open` statement where the criteria for the cursor filter is to be found. It is up to the programmer to choose which method to use.

CURSORS IN INFORMIX-4GL

A cursor is a data structure in the **front end** that references a data control structure in the database engine. Use of this data control structure in the engine requires that interprocess communication (IPC) facilities be used. Reduction of this IPC traffic is one means of optimizing cursor use (see Figure 3-2).

Figure 3-2 Cursors



Cursors cause activity to occur on both the front and back ends of the applicaton. When the cursor is created via the `declare` statement, data space is allocated in the application programs data area. This data space is a structure containing information about the cursor: the database object or objects it references, the table columns it retrieves, and any data binding that is to occur for parameters within the statements and data values to be retrieved by the statement.

When the cursor is opened the SQL statement is parsed, and any fatal errors are reported to the program. The database objects are examined for availability and, with the Informix cost-based optimizer, a number of access paths for the query are evaluated. An access path to the data is determined. If any temporary tables are needed, they are opened.

When data is retrieved from the database using the `foreach` or `fetch` statement, a certain amount of buffering will occur. Buffering occurs in the database engine and to a limited extent in the application. This buffer is usually around 1K in size and currently cannot be changed by the programmer.

TYPES OF CURSORS

There are four types of cursors in INFORMIX-4GL: the serial (or normal) cursor, the *scroll* cursor, the *update* cursor and the *insert* cursor. These cursors are explained below.

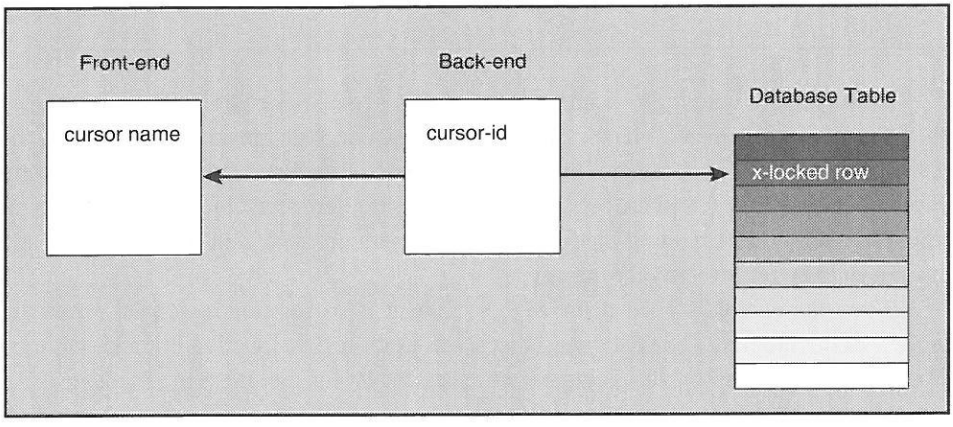
The **serial** cursor is the most common type of cursor. This cursor reads the selection set in a serial fashion, beginning at the start of the set and proceeding to the end of the set. This cursor cannot maneuver backwards and forwards through the set.

Whereas a serial cursor may directly access a table or set of tables through an index, a **scroll** cursor requires that a temporary table be created. Since the scroll cursor can step forwards and backwards through a set of data, it needs a snapshot of the data at a particular point in time. The movement can be forwards and backwards relative to the start of the set or to the current position in the set. Since a temporary table is required, some performance degradation can be expected when using this type of cursor.

SCROLL CURSOR

Figure 3-3 is an example of a scroll cursor. In this example, the user is allowed to scan the customer table. The user can move forward or back through the table, sometimes called browsing the table.

Figure 3-3 Scroll Cursors



The following code demonstrates the use of a scroll cursor. This application will display a screen form for the customer table and then allow the user to *browse* the table. The user will be able to move forward a record or back a record, jump to the start of the select set with the *first* option, or jump to the end of the select set with the *end* option. These movements within the current set will be made using a scroll cursor.

```
define customer_rec record like customer.*
define recno integer

open form f1 from "customer"
display form f1
```

```
-- select the key fields
declare c1 scroll cursor for
       select  *
       from customer

     open c1
     fetch c1 into customer_rec.*
     display customer_rec.* to customer.*

          menu "Customer "
                 command "Next" "Display next record"
                    fetch relative 1 c1 into customer_rec.*
                    display customer_rec.* to customer.*
                 command "Previous" "Display previous record"
                     fetch relative -1 c1 into customer_rec.*
                     display customer_rec.* to customer.*

                 command "First" "Display first record"
                   fetch first c1 into customer_rec.*
                   display customer_rec.* to customer.*

                 command "Last" "Display last record"
                   fetch last c1 into customer_rec.*
                   display customer_rec.* to customer.*

                 command "Goto" "Go to a specific record"
                     prompt "Enter record number: " for recno
                     fetch absolute recno c1 into customer_rec.*
                     display customer_rec.* to customer.*

                 command "Quit" "Quit this menu "
                     exit menu

           end menu
```

In this example, a `scroll cursor` is created for the customer record. The scroll cursor is then opened, and the first record is retrieved by the `fetch` statement. (By default, a `fetch` statement with no record number qualifier on a scroll cursor is considered a `fetch next` statement, a request to fetch the next record.) This record is then displayed to the screen.

Next, a menu is displayed, allowing the user to choose an option for *next*, *previous*, *first*, *last*, to go to a specific record, or *quit* the menu. For the *next* and *previous* options,

the fetch relative statement is executed with either a +1 for the *next* option or a -1 for the *previous* option. This will move the cursor to either the next record for the +1 or the previous record for the -1. This move will be **relative** to the current cursor position. Alternatively, the `fetch previous` statement could have been used to fetch the previous record, or the `fetch next` statement could have been used to fetch the next record. The *first* and *last* options simply execute either the `fetch first` statement to retrieve the first record or the `fetch last` statement to retrieve the last record.

The *goto* option allows the user to specify a record number in the current set and move to that record number. In this case, the move will be relative to the start of the set, so the `fetch absolute` statement is used. If it were desirable to move relative to the current position in the active set, then the fetch relative statement would have been used. Both of these statements take either an integer constant or an integer expression as an argument.

Problems with Scroll Cursors

There are two problems associated with some implementations of scroll cursors. Both problems are exhibited by the example shown above, and both are easily solved.

The first problem is that of stale data. When the scroll cursor is first executed, a temporary table is created; a snapshot is taken of the current state of the select set. This snapshot is then used for the duration of the cursor's existence. This means that as the user browses the rows in the set, it is possible that the data has been changed by another user.

The second problem is associated with the use of the temporary table. If the size of the selected tuple is large and the number of rows large, the temporary table could grow to a significant size. This could use a large amount of system resources and time to create.

The solution is to use two cursors: one to retrieve the primary key from the target table or tables, and the second to retrieve the rest of the data. The example below demonstrates this approach.

```
  define customer_rec record like customer.*
      define recno integer

  open form f1 from "customer"
      display form f1

-- select the key fields
declare c1 scroll cursor for
     select  customer_num
     from    customer

declare c2 scroll cursor for
    select *
    from customer
    where customer_num = customer_rec.cnum

open c1
```

```
open c2

fetch c1 into customer_rec.customer_num
fetch c2 into customer_rec.*

display customer_rec.* to customer.*


menu "Customer "

     command "Next" "Display next record"
       fetch relative 1 c1 into customer_rec.customer_num
       open c2
       fetch c2 into customer_rec.*
       display customer_rec.* to customer.*

     command "Previous" "Display previous record"
       fetch relative -1 c1 into customer_rec.customer_num
       open c2
       fetch c2 into customer_rec.*
       display customer_rec.* to customer.*

     command "First" "Display first record"
       fetch first c1 into customer_rec.customer_num
       open c2
       fetch c2 into customer_rec.*
       display customer_rec.* to customer.*

     command "Last" "Display last record"
       fetch last c1 into customer_rec.customer_num
       open c2
       fetch relative c2 into customer_rec.*
       display customer_rec.* to customer.*

     command "Goto" "Go to a specific record"
       prompt "Enter record number: " for recno
       fetch absolute recno c1 into customer_rec.customer_num
       open c2
       fetch c2 into customer_rec.*
       display customer_rec.* to customer.*

     command "Quit" "Quit this menu "
       let cont = "N"
       exit menu

end menu
```

In this example, two cursors are declared: one a scroll cursor and the other a simple cursor. As the scroll cursor is traversed for each row retrieving just the primary key, the remainder of the data record is retrieved from the database using the simple cursor. The data is then displayed to the screen.

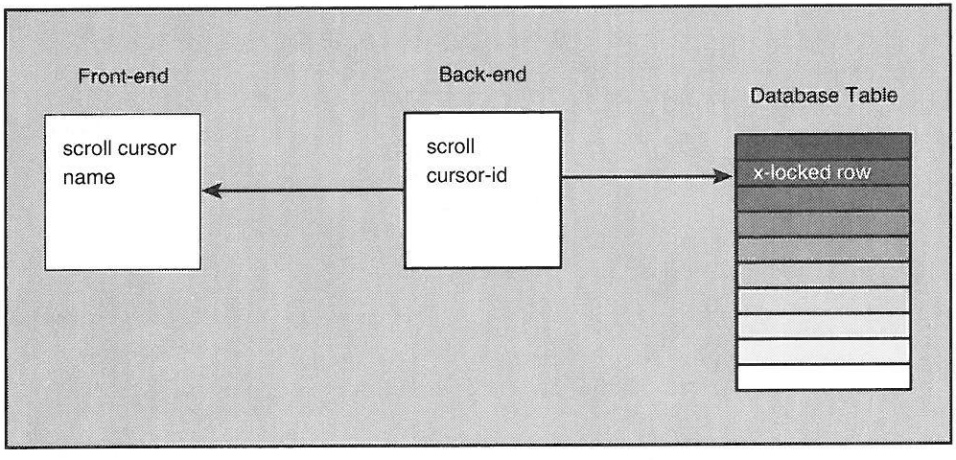
Update Cursors

An *update* cursor performs row retrieval similar to that of the serial cursor: it moves serially through the data set, retrieving rows. This cursor cannot move back rows and can only move forward one row at a time.

It is different from the serial cursor in two respects. First, it obtains a shared lock on each row as it is retrieved, and it releases the lock when it moves to the next row. Second, it has a number of restrictions associated with its use. These restrictions generally apply to the fact that, since it is intended to be used to update a single database table row, the underlying `select` statement can only reference one table and must reference the table directly. There cannot be an `order by` clause in the `select` statement, since this may require a sort table to be created and read instead of the target table.

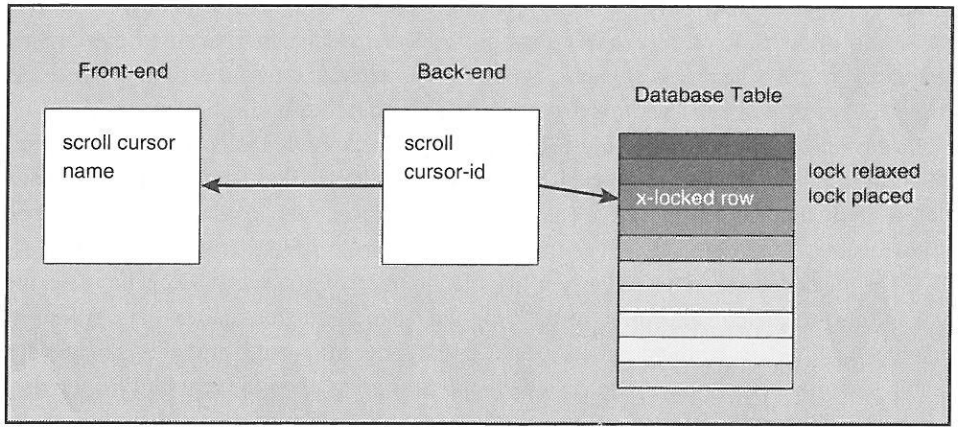
One very significant advantage to using update cursors is the performance advantage gained when performing database updates. Since the current cursor position can be used to update a record, the expense of finding the record to update is averted.

Figure 3-4 Update Cursor Locks Row



The update cursor is sometimes referred to as a *locking cursor*. As the update cursor accesses a row, it places a shared lock on the row (see Figure 3-4). A shared lock means that other users can read the row, but they cannot update the row until the user who acquired the shared lock has released the lock. As the cursor moves to the next row, the lock is released (see Figure 3-5).

Figure 3-5 Update Cursor Releases Lock

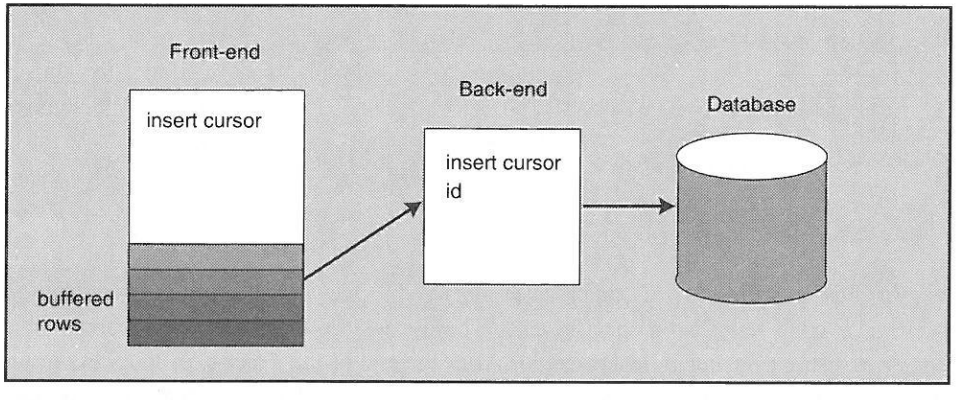


Insert Cursor

Unlike the other cursors mentioned, the *insert* cursor does not retrieve rows from the database. As the name implies, it is used to insert rows into the database. And unlike the simple `insert` statement, which inserts a single row into a database table, the insert cursor will buffer the rows before inserting them into the table, inserting multiple rows in a single insert request sent to the database engine (see Figure 3-6).

With the simple insert statement, a single row is sent through the IPC facility to the database engine where it is written to the database. When the row is sent to the engine by the front-end application, in this case an INFORMIX-4GL program, it waits for an aknowledgment from the engine that the row has been received. It is this aknowledgment and the time spent accessing the IPC facility that slows down the process of inserting multiple rows into the database and it is this step that is largely eliminated using an insert cursor.

Figure 3-6 Insert Cursor



With an insert cursor, rows are buffered in the front-end process before they are sent to the engine. The front-end buffer is usually about 1K in size, which for a 30-byte row would allow around 330 rows to be buffered before sending them to the engine. Instead of waiting for 330 separate acknowledgments for the rows, only a single acknowledgment will be issued for the entire set of buffered rows. For a process inserting 20,000 rows, only 60 buffers would be passed and 60 acknowledgments returned, versus 20,000 buffers and 20,000 acknowledgments using a simple insert statement. This yields a 98% reduction in pipe traffic and usually produces about a 40% performance improvement.

The insert cursor is useful when a large number of rows are being input to the database during a single execution of the program, such as a batch load of data. An insert cursor being used in conjunction in a data entry program would probably not yield a notable performance improvement. In a data entry program, the time required for the user to perform data entry offsets any possible performance improvement of buffering the rows. The rows will not make it to the database any faster, since the application must still wait for the user to enter the rows. An example of an implementation of the insert cursor is shown below.

```
define journal_rec record like journals.*

declare ins_curs cursor for insert into journal_hist⇐
     values ( journal_rec.* )
declare j_curs cursor for select * from journals where⇐
    journal_date < "1/1/1993"

open ins_curs

foreach  j_curs into journal_rec.*

     put ins_curs

end foreach

delete from journals where journal_date < "1/1/1993"

close ins_curs
```

In this example, two cursors are declared. One cursor reads rows from a `journals` table less than a specific date. These rows are then input to a history table (`journal_hist`) using an insert cursor. After the end of the `foreach` loop, the insert cursor is closed; this will flush the remaining rows in the buffer. Note that it is important

to close an insert cursor; otherwise, the rows remaining in the last buffer will not be output to the database.

Using an insert cursor also raises data integrity issues. Rows buffered in the **front end** do not come under the control of the database engine's data integrity facilities. Should the machine fail, these rows are lost. As was noted in the previous example, a significant number of rows could be lost this way. It is the programmer's responsibility to provide contingencies for this type of failure.

CURSOR IDENTIFIERS

Currently, INFORMIX-4GL cursors are represented internally with static module scope structures. These cursor identifiers are visible only within the source code module in which they were declared; any attempt to reference them in an another source code module would fail. Future versions of INFORMIX-4GL will overcome this limitation with global scope cursors and the ability to name cursors dynamically at runtime.

CLOSING THE CURSOR

The process of closing a cursor requires communication with the database engine with IPC facilities. The database engine must be informed of the **front end's** desire to close the cursor. If the cursor is successfully closed, the database engine returns an aknowledgement or response that the cursor has been closed or the process has failed. In an application where a cursor may be reopened several times as filter criteria change, performance can be improved by simply eliminating the cursor close operation.

This is allowed in INFORMIX-4GL because a cursor open implies a close and then an open of the same cursor. Using this technique will eliminate unnecessary IPC traffic and, depending on the specifics of the application, may improve performance significantly.

VIEWS

One of the most overlooked features of SQL is the `view`. A view is a name associated with an SQL statement; referencing the name is virtually identical to referencing the SQL statement. The following SQL statements provide an example.

```
create view  cust_profile
                ( cust_name,
                  cust_num,
                  cust_city,
                  cust_balance ) as
                select cust_name,
                       customer_num,
                       city,
```

```
                        balance
                from    customer, cust_balance
                where   customer.customer_num =
  cust_balance.cust_num
```

This SQL statement creates a view that relates to an SQL statement that joins the `customer` table with the `cust_balance` table. Once this view has been created, any references to `cust_profile` will execute the select statement shown below.

```
select *
from cust_profile
where  cust_num = 101
```

This statement will access the `cust_profile` view and execute the underlying query, appending the `where` clause criteria in the `select` statement above to the `where` clause criteria contained in the view. The SQL statement above is the equivalent of the SQL statement shown below.

```
                select cust_name,
                       customer_num,
                       city,
                       balance
                from   customer, cust_balance
                where  customer.customer_num =
  cust_balance.cust_num  and
                       customer.customer_num = 101
```

Obviously, using an SQL view, a great deal of the complexity of a query can be stored in the database, thus simplifying data access for the programmer. Reducing the difficulty and complexity of data access will help reduce the code required to develop an application, the overriding goal of any 4GL.

The SQL view can also be used to provide a limited access to a table. Making use of the SQL keyword `users` and a control table, a restricted view of any number of tables can be provided. The following example demonstrates this.

```
-- define a permissions table
create table user_perms ( logname char(20),
                    perm_level );

-- table to access
create table menu_header ( menu_id integer,
                     menu_name char(20),
```

```
                    perm_level );

-- create view to access menu_top table based on user login name

    create view menus as
       select  menu_name,
               menu_id
       from    menu_header, user_perms
       where   logname = user      and
               user_perms.perm_level = menu_header.perm_level
```

In this example, a table is first created that relates each user login name to a permission level. This table is then populated with user login names and permission levels (not shown). Next, a view is established that joins the `user_perms` table to the `menu_header` table using the `perm_level` column; this relationship is a 1-to-many relationship-for each `perm_level`, there are multiple `menu_header` entries in the `menu_header` table. (These `menu_header` table entries represent menus that this user is allowed to view.) Note that the view menus makes use of the SQL built-in keyword `user`, which resolves to the login name of the user executing the SQL statement. The result of the join being issued will be a restricted view of the entries in the `menu_header` table. All that must be done to extract this selected view is to execute the following query.

```
select *
from   menus
```

The view essentially represents the natural join of two sets: the `user_perms` and `menu_headers` tables. The view is a shorthand for this relationship. This set-oriented approach to data management is central to SQL. Since SQL is so tightly integrated into INFORMIX-4GL, the judicious use of this language will help ensure meaningful results when using INFORMIX-4GL.

MODULARITY AND SQL

In keeping with the overall goal of maintaining modularity in INFORMIX-4GL applications, SQL statements should be centralized and kept together in one or more source code modules. These modules shoud be devoted to similar operations, such as retrieving data or updating data. Additionally, for readability and maintainability, cursor or statement id declarations should be placed on a source code line close to where the cursor is to be used. The same holds true for statement ids.

An additional incentive for placing cursor declarations close to where they will be used in the code is in the INFORMIX-4GL requirement that cursor declarations precede

their usage in the source code; that is, a cursor must be declared on a previous line in the source code.

```
-- define a module scope variable as a flag
define first_pass smallint

function f()

let first_pass = TRUE

while ( continue = TRUE )
        call get_data( key1, key2 ) returning data_rec.*

...

function get_data( k1, k2 )
...

if ( first_pass )   then
  declare data_curs cursor for ...
             let first_pass = FALSE
end if
```

In the above example, the variable `first_pass` is declared to have module scope. Module scope variables are visible within the entire source code module in which they were declared and are static, retaining their value between function calls. These variables are not visible beyond the source code module in which they were declared. This technique overcomes the disadvantage of no static local variables. This type of variable is needed for the `first_pass` flag, since it must retain its value between calls to the function `get_data` something that simple local (automatic) variables will not do.

Within the function `get_data`, the `first_pass` flag is evaluated. If the flag is set TRUE, the initialization code within the `if/then` statement is executed; this includes the cursor declaration. If the flag is set FALSE, the initialization code is not executed. This allows the overhead of excessive cursor declaration to be avoided and enhances modularity by placing all code relevant to the cursor together in the same location.

Chapter 4

Screen Forms and INFORMIX-4GL

Screen Forms

The INFORMIX-4GL screen form is a template for screen interaction. It describes the data to be output to the screen, the characteristics of the data, and the attributes to be used on display of the data. A great deal of data binding can take place if desired. Screen fields can be bound to table columns; the data type and length of those columns will then match the data type and length of those columns when the form was created.

INFORMIX-4GL uses a form compiler, named `form4gl`, which reads the screen form image and outputs a compiled image of the screen form. The data in this file maps to internal structures that are used by the INFORMIX-4GL program at runtime. The INFORMIX-4GL screen form interface is designed to be dynamic.

The characteristics of the terminal are read at runtime from the `termcap` or `terminfo` definition file. This means that, by definition, specific terminal characteristics are not known until runtime. Therefore, the compiler cannot determine the validity of certain screen form interaction statements. This can make the screen form development process frustrating. Some tips to make this process easier and more predictable will be presented in this chapter.

Managing Screen Size

The size of the screen form defaults to 24 lines at 80 characters per line, but this can be changed by using the `screen size` statement in the screen form. Since different terminals can have different screen sizes, the validity of these parameters is not checked until runtime. As long as the terminal capabilities match those needed to run the application, the application will run.

Screen Form Programming

The INFORMIX-4GL screen form is composed of a `database` section, a `screen` section, a `tables` section, an `attributes` section and an optional `instructions` section. The `screen` section contains an image of the screen. The image is composed of plain text and screen *fields*. Screen fields are labels that

indicate where input or display is to take place. These fields are identified by screen labels, which are composed of an open brace, an initial alpha character and/or additional alpha characters and numbers. After the label, the field is padded by spaces up to the defined length of the field. The end of the field is identified by a closing brace. Alpha characters used in the screen field labels can be upper or lower case; the screen form compiler is not case-sensitive. This implies that only a single character can be used to create a field label for a single character field. The maximum number of single character fields that can be placed in a screen form is therefore 26.

DATABASE BINDING IN SCREEN FORMS

Since INFORMIX-4GL screen forms were designed to interact with a database table or tables, a form of database binding is available within the screen form interface. Once the database has been identified in the `database` section of the screen form, database table fields can be bound to fields on the screen form in the `attributes` section. The data type of these fields is then presumed to be the data type of the field in the database table. The length of the field, however, is the length of the field on the screen form. At runtime, when data is input into these fields, its data type is checked against the data type of the database table field. If the data that has been input cannot be converted to the data type of the table field, an error is signaled and user input remains in the same field. If the data length differs and truncation does not cause a loss of significant digits, truncation will occur. In this case, no error is signaled.

The `attributes` section further provides the means of specifying additional attributes for the screen form field. Most of these attributes can be specified within the INFORMIX-4GL application as well as in the screen forms. If they are specified in both places, some contention can occur.

The `instructions` section of the screen form is optional. It can be used to redefine the field delimiter characters used in INFORMIX-4GL or create screen records. Screen records are a useful means of grouping screen form fields together in a single record which can be referenced in INFORMIX-4GL screen interaction statements.

`formonly` SCREENS

If the screen form will not be interacting with a specific database, the database statement can specify `formonly` as the database. In this case, the `tables` section cannot be used.

The `formonly` screen form provides an excellent mechanism for designing generic screens—screens not bound to a particular database. These screens can be used to develop library routines usable for manipulating data from a variety of tables. Such routines are the building blocks of a modular application.

A formonly screen is designated by indicating in the database section of the form that no database will be used. The syntax for this is shown below.

```
database formonly
screen {

...
```

Of course, designating a form as formonly precludes using any database binding within the form. Using database binding does not limit the screen to formonly fields. As the form below shows, it is possible to mix fields bound to table columns with formonly fields.

```
database inventory
screen
{

Product Name     [f001                ]
Wholesale Price  [f002         ]
Markup           [f003         ]
Total            [f004         ]

}

tables
inventory
stock

attributes
f001 = stock.product_name;
f002 = stock.curr_price;
f003 = inventory.markup_amt;
f004 = formonly.total;
...
```

In this example, the inventory database is selected because the majority of the fields in the form are from tables in that database. However, 1 of the fields, the total field, is a computed field; the data in this field does not come from any database tables. For this reason, the field is designated as a formonly field.

Since there is no data type declared for the formonly field, it will be treated as a character field with a length equal to the length of the screen form field in the screen form including blanks. Even though a numeric will be displayed to this field, there should be

no problem since data conversion will occur freely as it does in most INFORMIX-4GL statements. A data type could have optionally been declared for this field using the following syntax.

```
....

attributes
f004 = formonly.total, type decimal(5,2);
```

MANIPULATING THE SCREEN FORM IN INFORMIX-4GL

INFORMIX-4GL manipulates data on the screen via the screen form. Screen forms can be displayed in a window, and multiple windows and screens can be opened and displayed at the same time. The only real restriction on the number of screen forms opened is the number of file descriptors available on the application platform, since each open screen form represents an open file.

The `open form` statement is used to open a screen form. This statement is used to associate a screen form identifier with a screen form. It also identifies where the runtime application can find the compiled screen form file necessary to open the screen form. The validity of this information is not checked until runtime.

Screen Form Identifiers

The screen form is identified by an identifier that has global scope in the application. The identifier is static and is created at *compile* time; it cannot be created at *runtime*. This implies that a form opened in 1 INFORMIX-4GL source code module will be recognized in another source code module.

As with all global identifiers, this can create problems. A screen form that is opened, used and closed within the same function could have a name collision with a screen form opened and used in an unrelated function. One workaround for this problem is to name generic screen forms using a leading underscore character, as shown below.

```
open form _custf1 from "cust_form"
display form _custf1
```

MODULARITY AND SCREEN FORM INTERACTION

When developing an application, there are several decisions that must be made relative to screen forms. One programming style would elect to open all screen forms in the same function. These forms are opened during application initialization and are then ready for use. This eliminates the need to open the screen form when the application finds

it necessary to use it; the form will already be opened—all that will be necessary is to display the screen form.

There is some minimal overhead associated with opening a screen form. The screen form file must be opened, which requires operating system overhead. The contents of the file must be read into INFORMIX-4GL's internal screen form structures; data space must be allocated for these structures. While this may have been an issue on some older, less efficient platforms, most current hardware performs these functions in a virtually imperceptible period of time.

A coding style that enhances modularity and improves maintainability is to place all INFORMIX-4GL screen form operations together in 1 or more screen form modules. In this module, screen forms are opened, displayed and, when necessary, closed.

An alternative to opening screen forms together in a single function designated solely for opening screen forms is to open the screen form in the function in which it will be displayed and used. The advantage to this approach is maintainability. The programmer need look no further than the current function to determine which screen form is being used to display the screen.

Managing Multiple Windows

It has become something of an art to determine the correct window size for an INFORMIX-4GL screen form. A window can be opened with a screen form, but the default window size is not always that needed for the application. The following formula can be used to size a window for an INFORMIX-4GL form.

number of lines needed = (form line - 1) + (form length + 1 for the comment line)
+ 2 (if the window has a border)
number of columns needed = position of right-most column + 2
(if the window has a border)

By default, the form line starts at line 1 of the screen. This can be changed with the `options` statement. As the formula shown above indicates, changing the form line can impact the sizing of the screen.

The Current Window

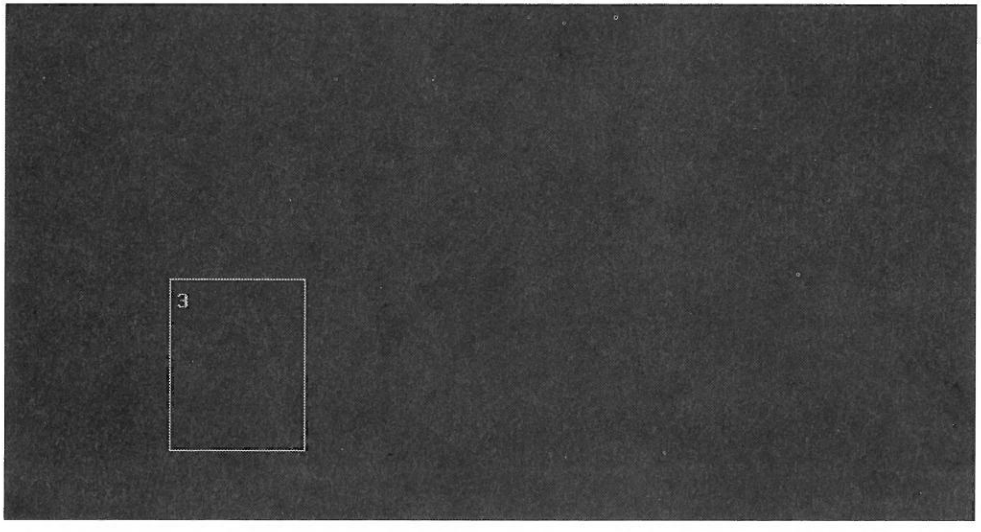
INFORMIX-4GL allows multiple screens to be opened and displayed. Control can be passed from 1 window to another, with each window displaying separate screen forms and performing separate input operations.

The `current window` statement is used to shift control between windows. When the statement is executed as `current window is screen`, the effect is to have the bottom-layer window, the blank screen, brought to the top and all other windows layered below it. Once this statement has been executed, any displayed windows or forms

will be displayed on this blank screen. The program below demonstrates this functionality.

```
main

open window w1 at 5,5
   with 10 rows, 10 columns
   attribute (border )
display "1" at 1,1

current window is screen
open window w2 at 10,10
   with 10 rows, 10 columns
   attribute(border)
display "2" at 1,1

current window is screen
open window w3 at 15,15
   with 7 rows, 10 columns
   attribute( border )
display "3" at 1,1

sleep 10

end main
```

Figure 4-1 Managing INFORMIX-4GL Windows.

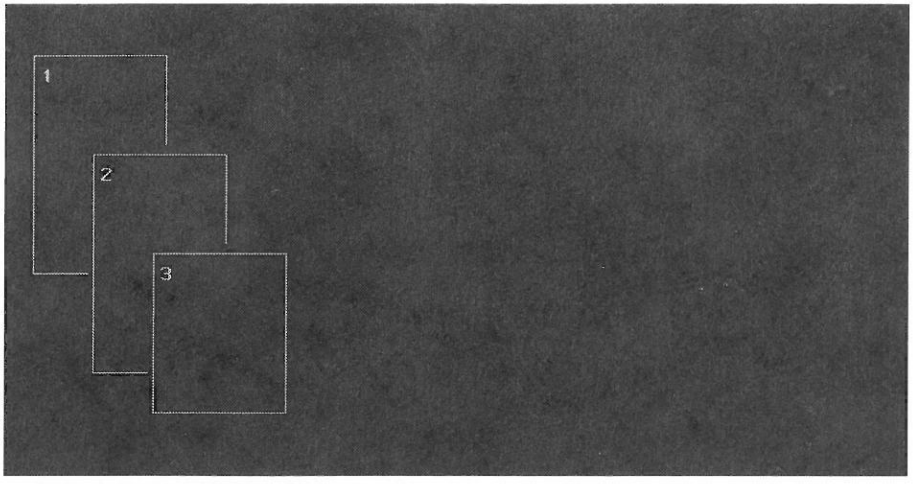


This program displays three-bordered windows to the screen, each with a separate number displayed to the window. But as each window is displayed, the `current window is screen` statement is executed and eclipses the previous window with a blank screen. The result is, after the last window has been displayed, that it is the only window visible, as shown in Figure 4-1.

To obtain a set of layered windows, the `current window's screen` statement must be eliminated as shown in the code below. The result is a set of three layered windows.

```
main

open window w1 at 5,5
   with 10 rows, 10 columns
   attribute (border )
display "1" at 1,1

open window w2 at 10,10
   with 10 rows, 10 columns
   attribute(border)
display "2" at 1,1

open window w3 at 15,15
   with 7 rows, 10 columns
   attribute( border )
display "3" at 1,1

sleep 10

end main
```

Figure 4-2 Managing INFORMIX-4GL Windows



In this code, each window is displayed sequentially. The result is that the first window displayed is the bottom layer, the next window displayed is the second layer and the third and last window displayed is the top layer. The result is shown in Figure 4-2.

Unfortunately, there is no easy way to determine the current window. This makes it advantageous to keep window manipulation statements close to the code in which the windows are used, preferably within the same function block. If this maxim is followed, it will be relatively easy to determine which window is the current window merely by looking to the start of the current function. The following code sample illustrates this concept.

define wflags array[3] of smallint

```
main

initialize wflags to NULL

call f1()
call f2()
call f3()

end main

function f1()

if wflags[1]  IS NULL then
```

```
    open window w1
    open form f1
    let wflags[1] = TRUE
end if
current window w1

...

end function

function f2()

if wflags[2] IS NULL then
   open window w2
   open form f2
   let wflags[2] = TRUE
end if
current window w2

...

end function

function f3()

if wflags[3] IS NULL then
    open window w3
    open form f3
    let wflags[3] = TRUE
end if
current window w3

....

end function

function close_windows()
define n smallint

for n = 1 to 3

    if wflags[n] is NOT NULL  then
        case n
```

```
            when 1 close window w1
                   close form f1
            when 2 close window w2
                   close form f2
            when 3 close window w3
                   close form f3
        end case
    end if

end for

end function
```

This sample application demonstrates a modular approach to window and form management. First, an array of three `smallint` variables (`wflags`) is declared with module scope. These variables will be used as flags in the functions that will make use of the screen forms. In the main program block, the array of flags is initialized to NULL.

Next, the functions are called in sequence. In the body of each function, the flag variable is tested; if it is NULL, the window and screen form for that function is opened and the screen form displayed. If the flag is not set, the current window is set to the appropriate window and function execution continues. The `close_windows` function is used to close only those windows that have been opened; it can determine this information by evaluating the `wflags` array element for the window in question.

This approach to screen window operations modularity places the majority of window operations in the functions that use the window, thus keeping the control for these operations close together. The advantages to this approach are improved modularity and efficiency; windows and forms are only opened if they are used.

`display` STATEMENT

The `display` statement is used to display output either to the current screen or to specific fields on the current screen form. Since there is no supported method of determining the current screen form, it is the programmer's responsibility to have knowledge of the current open screen form.

The most simple form of the `display` statement can be used to perform simple output of data to the current position on the terminal screen. When this form of the statement is used, there is no effort to clear or initialize the current terminal screen. The statement acts like the C language `puts` function call; it simply takes the data variables passed to it, converts them to a string data type, and outputs the data at the current terminal cursor position. This means that in its simplest form, the `display` statement can be used to produce simple reports, as the following code demonstrates.

```
main
define salesm record like salesman.*

declare sales_curs cursor for
                   select *
                   from  salesman
           where region  = "NE"

foreach sales_curs into salesm.*
       display "Salesman:  First Name: ",
               salesm.first_name,
               " Last ", salesm.last_name,
               " Phone: ", salesm.phone

        end foreach

end main
```

In this example, a report of all the salespersons in the northeast region is printed to the terminal screen. Since no screen interaction statements that would reset the terminal screen have been executed, the display statement will simply take its output and send it to the terminal screen, displaying a newline character at the end of the output. The output of the program could be output to the system printer using the UNIX `lp` command as follows.

```
rpt_prog | lp
```

The simple form of the display statement can take any INFORMIX-4GL simple data variable and output it to the screen. Unfortunately, in its current form it cannot take expressions as arguments, even if those expressions evaluate to a simple data type. The example below illustrates this problem.

```
function funct_s()

  return  "hello world"

end function

display funct_s()
```

This code example would fail to compile, since `funct_s` is considered too complex a data type for the `display` statement, even though the expression

`funct_s()` resolves to a simple character string. The following statement would also fail to pass a compile.

```
define x,y smallint

let x = 1
let y = 2

  display ( x + y / 2 )
```

Once again, the compiler would consider the simple mathematical expression too complex for the `display` statement. The only solution to this limitation is to use temporary variables to store the results of these expressions, as shown below.

```
let str = s()
let x = x + y / 2

display x,str
```

This `display` statement would not only compile, but would also execute without incident.

The `display` Statement and Screen Form Interaction

The `display` statement can make explicit use of database binding with the `by name` clause. When this clause is used, the INFORMIX-4GL runtime screen form component will search for column names that match the record component names of the record being used. These record component names must provide an exact match for the screen form field names; otherwise, the statement will fail. An example of this statement is shown below.

```
database sales
screen
{
                                    Customer Record

      First Name        [f001                              ]
      Last Name         [f002                              ]
      Address           [f003                              ]

                  }
tables
customer
```

```
attributes
f001 = customer.first_name;
f002 = customer.last_name;
f003 = customer.address;

....

-- INFORMIX-4GL program

    define cust_rec record like customer.*

...
    display by name cust_rec.* to customer.*
```

In this example, since the fields in the screen form are named identically and in the same order as the columns in the customer table, they can be referenced in the `display` statement as shown above.

It is not necessary for an entire table to be referenced for the `by name` clause to be used; a portion of 1 table or portions of a number of tables could be referenced. As long as the *names* of the columns are identical to the record element name and sensible data type conversion can take place, the `by name` clause can be used. The example below demonstrates this capability using a screen record.

```
database sales
screen
{
                          Customer   Record

        First Name      [f001                          ]
        Last Name       [f002                          ]
        Salesman Numver [f003                          ]

                }
tables
customer
salesman

attributes
f001 = customer.first_name;
f002 = customer.last_name;
f004 = salesman.number;
```

```
instructions

screen record scr1 ( customer.first_name,
                     customer.last_name,
                     salesman.number)

...

--- INFORMIX-4GL program to display data to the screen form

      define scr_rec record like customer.first_name,
                                 customer.last_name,
                                 salesman.first_name,
                                 salesman.last_name

....

      display by name scr_rec.* to scr1.*
```

In the example above, the screen form is created to interact with two tables: the `salesman` table and the `customer` table. The columns used in these two tables are identified and referenced in the `attributes` section of the screen form. These columns are then placed in a screen record. In the INFORMIX-4GL program, this screen record is then used as the target of the `display` statement. The entire contents of the `scr_rec` record will be displayed to the `scr1` screen record, matching the names of the record elements to the names of the columns.

input STATEMENT

The `input` statement uses the INFORMIX-4GL screen form to retrieve data from the terminal. While the majority of the data entry control rests with the INFORMIX-4GL program, the screen form definitions also affect data entry. The INFORMIX-4GL program interacts with the screen form to retrieve data from the terminal. This interaction is performed using the `input` statement.

The `input` statement retrieves data from each field in its target form. It moves through the fields in a serial fashion, beginning at the top of its internal input list and finishing with the last field in its input list. The order of input is controlled by the order of fields listed in the `attributes` section of the screen form if a screen record is **not** being used, or the order of fields listed in the screen record if a screen record is used.

```
define cust_rec record like customer.*
```

```
-- the order in which the fields appear in the screen
-- form will control the order of input

input cust_rec.* from customer.*
```

A number of events can be trapped during the execution of the `input` statement. These event traps are 1 of the more unique and interesting aspects of the INFORMIX-4GL language. The events are *before field*, *after field* and *on key*. The `before field` statement is used to take control of the input operation before entering the named field. The `after field` statement takes control after the cursor leaves the field. The `on key` statement is used to take control of the application when a defined function key is pressed by the user. Examples of these statements are shown in the next section and in the code presented in Chapters 7 and 8.

Moving Past an Input Field

There are times when it is desirable to keep the cursor from entering a field on the screen form. The field may be used to display lookup information or a field derived from other fields on the screen form. There are two methods for skipping this field. In the code sample below, the screen field is skipped using a `next field` statement in the `before field` block.

```
database sales
screen
{

    Salesman Information

First Name          [f001                          ]
Last Name           [f002                          ]
Territory Code      [f003   ]
Name                [f004                          ]
}

tables
salesman
territory

attributes
f001 = salesman.first_name;
f002 = salesman.last_name;
f003 = salesman.territory;
f004 = territory.territory_name, reverse;
```

```
instructions

screen record sc1 (salesman.first_name,
salesman.last_name, salesman.territory,
territory.territory_name)

...

define scr_rec record like salesman.first_name,
salesman.lastname, salesman.territory,
territory.territory_name;

input scr_rec.* from  sc1.*

      before field territory_name
            display scr_rec.territory_name to
                    territory_name_
            next field salesman.first_name
...
```

In this example, the `input` statement will skip the `territory_name` field because the `before field` statement for this field directs input to the `salesman.first_name` field before the cursor can enter the `territory_name` field. This same functionality could have been provided by declaring the screen field attribute in the screen form to be a `noentry` field as shown below.

```
...
f004 = territory.territory_name, reverse, noentry;
```

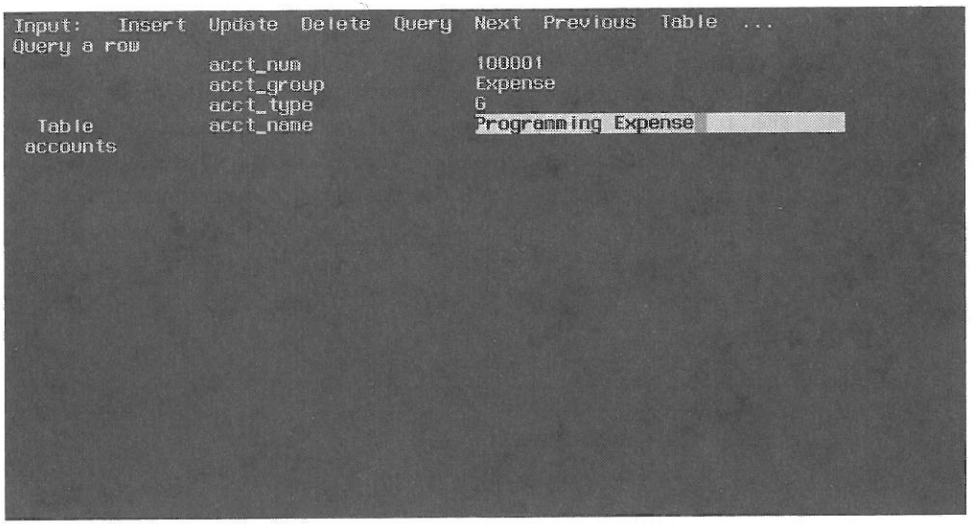
Either approach is correct. Placing attribute entries in the screen form has the advantage of reducing the code required in the INFORMIX-4GL application. However, a maintenance programmer reviewing the input loop of a given application would not immediately know what attributes are being used on the screen form fields. By placing attribute information within the `input` statement block, the programmer uses a structured syntax to achieve the same functionality.

Highlighting the Current Field

Often, it is desirable to have the current input field highlighted as shown in Figure 4-3. This makes it easier for the user to detect which field is currently being input. Unfortunately, INFORMIX-4GL does not currently have a clause which triggers this

behavior. However, as the following example shows, it is not difficult to achieve this functionality.

Figure 4-3 Highlighting the Current Field




```
-- highlight the current field in the input loop

input scr_rec.* from scr1.*

      before field first_name
          display scr_rec.first_name to first_name
                attribute ( reverse )

      after field first_name
          display scr_rec.first_name to first_name
                attribute ( normal )

      before field last_name
          display scr_rec.last_name to last_name
                attribute( reverse )

      after field last_name
          display scr_rec.last_name to last_name
                attribute( normal )
```

In this example, the current field is highlighted in the `before field` statement by displaying the data in that field to the screen using the `attribute (reverse)` clause. This will leave the entire field (length defined in the screen form) in the reverse attribute while the user is entering data and the cursor remains in this field. In order to eliminate the reverse video when the cursor leaves the field, the code in the `after field` statement is required. This code will display the current contents of the data record, which may have changed during input, back to the screen field using the *normal* video attribute. The `attribute` statement in this `display` statement is redundant since normal attribute is the default attribute for the `display` statement, but it is included here to make the code more readable. A maintenance programmer would be more likely to understand the purpose of the `display` statement because the normal attribute is being emphasized by use of this clause.

TERMCAP ISSUES

A number of the capabilities of INFORMIX-4GL are dependent on the termcap definitions of the terminal being used. If a terminal attribute is not available and INFORMIX-4GL attempts to invoke this attribute, the statement will generally be ignored by the screen form runtime component.

For instance, if the terminal does not support reverse video, reverse video will simply not be displayed. Additionally, if the function keys are not assigned in the termcap for the terminal, INFORMIX-4GL will fail on attempts to access these function keys.

Redefining Function Keys

INFORMIX-4GL allows specific keys to be remapped through the `options` statement. If a programmer wishes to turn off certain keys during input, the `options` statement may be used to redefine these keys to a function key that is undefined for the terminal being used.

Turning Off the Insert Key and Delete Key in `input array`

If an application uses an `input array` statement that should not allow the user to insert or delete rows, the `options` statement can be used to remap the insert or delete keys to function keys undefined for the terminals being used, as shown below.

```
options
  insert key f39,
  delete key f40
```

This code will redefine the `insert key` to function key 39 and the `delete key` to function key 40, keys that cannot be generated on most terminals. If the terminal

doesn't allow these function keys, the `insert key` and `delete key` have been turned off during the `input array`.

Adding Additional Function Keys in the Termcap File

Adding additional function keys to the INFORMIX-4GL `termcap` is relatively simple. The programmer must first determine the escape sequence transmitted by the terminal. This can usually be accomplished by pressing the function key and redirecting its output to a file. This file can then be read using either a text editor that will display control keys or a UNIX utility, such as `od`, that will display the ASCII or hexadecimal value of the contents of the file.

Once the function key escape sequence has been determined, it is simply a matter of making the appropriate entry in the `termcap` file. For example, if the terminal being used was found to be sending the escape sequence \E (escape) OP for function key 1 (f1), the correct entry to make in the termcap for that terminal would be as follows.

```
v1|vt100|vt-100|DEC vt100:\
...
        :k0=\EOP:k1=\EOQ:k2=\EOR:k3=\EOS:\
...
```

If a *page-up* or *page-down* key is needed, then the process is similar to that of the function keys. The programmer must first determine the escape sequence transmitted. Once this is known, the termcap entry must be made to relate this escape key sequence to a function key number that will be used to trap the page-up and page-down keys. This process is shown below.

```
v1|vt100|vt-100|DEC vt100:\
...
        :k31=\EOP:k32=\EOQ:k2=\EOR:k3=\EOS:\
...
```

Next, in the `options` statement for the application, the function keys for next key and previous key should be set to the function keys that have been mapped for the page-up and page-down keys, as shown below.

```
options
  next key f31,
  previous key f32
```

Once these steps have been taken, whenever an `input array` or `display array` statement is executed, the page-up and page-down keys for the terminal will cause the screen array to scroll forward or backward appropriately.

TRAPPING THE INTERRUPT KEY

INFORMIX-4GL provides a means of trapping the interrupt key when the application is waiting for user input. This interrupt trap is achieved using the `defer interrupt` statement. By default, if the interrupt signal is received by the program, the application will abort. Since this is not always desirable behavior, it is useful to trap the interrupt key in the application. If the interrupt key is trapped, the interrupt flag (`int_flag`) must be checked at the end of `input` statements to determine whether or not the user has pressed the interrupt key. The following code demonstrates this method.

```
database accounting

main
define ar record like accounts.*

defer interrupt

options  message line last - 1

open form accts from "accts"
display form accts

let int_flag = FALSE
input by name ar.*

   before field acct_num
      message "Enter Account Number"
      display ar.acct_num to acct_num attribute( normal )

end input
if int_flag  then
   error "Input Aborted.
   initialize ar.* to NULL
   return ar.*
else
   return ar.*
end if
```

In this example, the `defer interrupt` option is set before the screen form is opened and displayed. Before the start of the input loop, the `int_flag` is set to FALSE. This is necessary to allow for the contingency when the `int_flag` has not been reset by the application after the last interrupt.

After the input block, the `int_flag` is tested. If it has been set, an error message is displayed, the `int_flag` is reset, the `ar` record is set to NULL and the function returns a NULL record. If the `int_flag` has not been set, the function simply returns the record.

Chapter 5

INFORMIX-4GL Reports

In creating the INFORMIX-4GL report writer, developers sought to provide a productive report generation environment that would overcome the limitations of Ace, the precursor to the INFORMIX-4GL report writer. The syntax of Ace was placed in the INFORMIX-4GL report writer virtually intact, and was combined with virtually all of the .4GL procedural and nonprocedural statements. The result was the INFORMIX-4GL report writer.

The INFORMIX-4GL report writer provides a block-oriented, structured report development environment. Control blocks are provided to define format, initialize control variables and produce report output. Within this structure, calculations can be performed, database tables updated and output formatted. Tables 5-1 and 5-2 list the statements that make up the INFORMIX-4GL report facility.

Table 5-1 Report Statements

Statement	Function
`report report_name( parameter1, parameter2, ...)`	Indicates the start of the report block
`define`	Defines static variables local to the report
`format`	Indicates the start of the block that defines how to format the data being passed to the report
`on every row`	Within the format section—indicates how data is to be formatted for every row passed
`after group of / before group of field_name`	Within format section—indicates control breaks for the data stream being passed to the report
`on last row`	Within format section—indicates actions taken at the end of the report

Virtually all other INFORMIX-4GL statements can be executed in the `format` block of a report function with few restrictions but some restrictions do apply. Screen interaction statements such as `close form`, `input`, `display` and `construct`

cannot be executed within the body of a report. The simple workaround for these restrictions is to call a function, which will then execute the screen interaction statement.

REPORT STATEMENTS

A report has a specific start triggered by a `start report` statement, and a specific end triggered by a `finish report` statement. The `start report` statement opens the file or device for output. The `output to report` statement is used to output data to the file or device; it is called for each tuple to be output to the report. The `finish report` statement flushes the remainder of the buffer being used to the output device and closes the device. Failure to issue a `finish report` statement could result in an incomplete report (a common error). The effect of these statements is shown below.

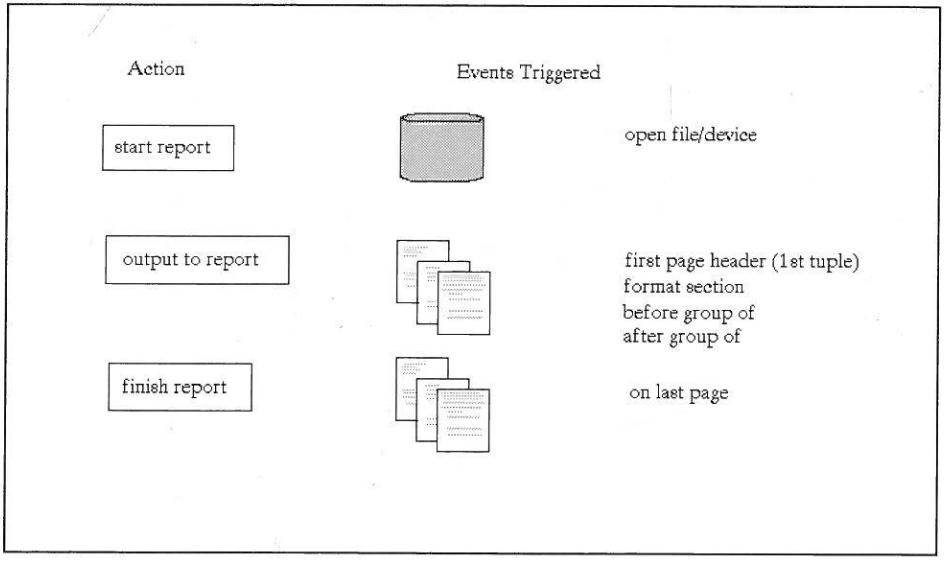
Table 5-2 Report Interaction Statements

Statement	Function
`start report <report name>`	Starts the report; can optionally identify an output device
`output to report <report name>`	Sends output to the report device/file
`finish report <report name>`	Completes the report; closes the output device

Variables are declared within the `define` section and, unlike other INFORMIX-4GL function variables, these variables retain their value between successive calls. This is currently the only instance of static local scope variables within INFORMIX-4GL.

The report block is declared with the `report` statement. It uses a function-like syntax which identifies the parameters to be passed to the report using the `output to report` statement on each iteration of the report. These parameters must be passed to the report, and must match in number and ordinal position and have compatible data types. If the number of parameters in the `output to report` statement does not match that declared in the report declaration statement, the **compiler** will signal a fatal error. If the data types passed in the `output to report` statement cannot be converted, a fatal **runtime** error is signaled. Figure 5-1 illustrates the effects of report statements.

Figure 5-1 Effect of Report Statements



Declaring a Report

The body of a report is declared using a syntax similar to that of an INFORMIX-4GL function, as shown below.

```
report sales( sales_rec )
define sales_rec record like sales_person.*
```

This code declares the report `sales_report` and indicates that it will be receiving the `sales_rec` record. In the `define` section of the report, `sales_rec` is identified as a record with a structure identical to that of the `sales_person` table.

The report declaration need not appear in the same module in which the report is started and executed. Internally, the report declaration creates a function that will be called when the `output to report` statement is executed, as follows.

```
start report sales_report

foreach report_curs into srec.*

     output to report  sales_report ( srec.* )
```

```
end foreach

finish report sales_report
```

In this example, the `output to report` statement is executed within a `foreach` loop. The `foreach` loop provides a excellent mechanism for report generation. It creates a loop within which report parameters can be created and then output to the report, all within the same control block. Before the start of the `foreach` loop, the `start report` statement can be executed; upon termination of the `foreach` loop, the `finish report` statement is usually executed.

Generating a Default Report

INFORMIX-4GL has the capability to create a simple default report using very little code. The report produced is very simple and lacks formatting, but it does represent a quick and easy method to produce a report. The following code demonstrates this capability.

```
database sales

main
define sr record like sales_person.*

declare c1 cursor for
    select *
    from sales_person

start report sales

foreach c1 into sr.*
     output to report sales( sr.* )
end foreach

finish report sales

end main

report sales( sales_rec )
define sales_rec like sales_person.*

format
   every row
```

```
end report
```

In the above example, on the first line of the program module, the compiler is informed of the database to use for record declarations, the `sales` database. Next, a record is declared to have the same structure as the `sales_person` table. A cursor is declared to select every row, and every data element in the rows, from that table. A `foreach` loop is then started, and the data retrieved is output to the default report. The output from this report is shown below.

```
first_name    Frank
last_name     Salesguy
territory     Buffalo
quota         1,000,000

first_name    Greg
last_name     Techguy
territory     SW
quota         1,000,000
```

The `output` Section

The `output` section of the report, as the name denotes, defines where data will be output and provides some basic formatting instructions for that output. This section of the report is optional and, if omitted, various defaults will take effect. The defaults for the report output section are listed in Table 5-3.

Table 5-3 Output Section Defaults

Parameter	Default
`report to <character>`	Terminal screen (stdout)
`left margin <integer>`	5
`right margin <integer>`	132
`top margin <integer>`	3
`bottom margin <integer>`	3
`page length <integer>`	66
`top of page <character>`	Not used

The parameters for these options must be constants; variables may not be substituted. While this may appear to force the report format options to be defined at

program compile time, the fact is that these parameters are easy to adjust dynamically at runtime, as shown in the sections below.

The `report to` statement is used to direct the output of a report to a particular location. Note that by default, the report data will be sent to the terminal screen in a continuous stream. Unless `pause` statements are inserted into the body of the report, the report will scroll continuously off the screen until the report is complete.

If the `report to` statement has been used in the body of the report, and the `start report` statement has a `to` clause, then the `start report` statement takes precedence, and the `report to` statement in the body of the report is ignored.

The `report to` statement can specify `printer` as the output device. In this case, the contents of the environment variable DBPRINT are examined, and output is sent to the program specified in the environment variable. If this environment variable is not set, then, under the UNIX operating system, the program "lp" is opened as a receiving pipe, and data is output to this program.

The `right margin` parameter in the output section is only effective within a default report or if the `wordwrap` clause is used; if this is not the case, the `right margin` statement is **ignored** by the INFORMIX-4GL report writer. This means that, in most cases, INFORMIX-4GL does not enforce the right margin; it is up to the programmer to do that, using logic similar to that shown below.

The `order by` Section

The `order by` statement is used to specify the order in which the report grouping statements (`before group of`/`after group of`) should be executed. If the sort order of the report is specified here, the report *front end* will sort the data. Informix has made a great deal of effort to improve the speed of engine or *back end* sorts. The back end has the intelligence to make use of indices if available, use parallel sorting schemes or sort merge techniques, methods that are not necessarily available to the front-end sort facility. If the data will be output to the report in the order expected, the order need not be specified.

Since it would obviously be more efficient to have the data sorted by the engine than by the front end, it is recommended that sorts be performed there. This can be done by specifying the correct sort order using the `order by` clause in the SQL statement.

The `format` Section

The `format` section represents the body of the report. Subclauses within this section identify the actions to be taken during the preparation of the report. It is the capability of using virtually any INFORMIX-4GL procedural or nonprocedural statement within this block that allows the INFORMIX-4GL report writer to be used for more than just simple reports.

FIRST PAGE HEADER

The `first page header` section is executed once per report. It is triggered by the transfer of the first tuple to the report function. The `first page header` does **not** trigger the `page header` block, so if it is necessary to print a page header for the first page, the page header print statements must also appear in the `first page header` section of the report. This section is usually used to print the first page header, but is also useful for performing initialization of variables to be used in the report, as the code below demonstrates.

```
define

    col1 integer,
    col2 integer,
    col3 integer,
    page_header1 char(50),
    page_header2 char(50)


first page header

-- initialize variables
let col1 = 10
let col2 = 20
let col3 = 30

-- create the page header strings to be printed on every page
let page_header1 = "Regional Sales Report"
let page_header2 = "Date: ",today, " Time: ",time

-- print the page header
print  col1, page_header1
print  col1, page_header2
...
```

In this example, the `first page header` section is used to initialize integer variables that will be used to hold the multiple column position for the data to be printed on the report. Next, two character strings are initialized to hold the page headers for the report. Finally, the page headers are printed.

This demonstrates initialization of variables within the `first page header` section of the report. These variables are local to the report function and are not visible outside of the report function. However, unlike other local variables in INFORMIX-4GL, they will retain their value on each iteration of the report. This allows subtotals and totals

to be maintained and allows local report variables to be used to hold column positions, which is demonstrated below.

Creating Dynamic Column Positions in Reports

It is not uncommon during the development of a report to have to constantly change the format of the report. By using the integer variables to hold column positions, report modification is made easier.

If all column one data is being printed at a column position stored in an integer variable, it is merely a matter of changing the value of that integer variable once in the `first page header` section to effectively move the data throughout the report. Storing the column position in this manner also allows the data to be moved around on the report at runtime if needed.

A variation on this method is shown below. With this method, all column positions are relative to the starting column (column one) position. In this way, adjusting the column position for one column will automatically adjust the column position of the remaining columns to the right of that column. This technique is shown below.

```
first page header

   let col1 = 3
   let col2 = col1 + 10    -- sales man first name
   let col3 = col2 + 5     -- sales man last name
   let col4 = col3 + 5     -- region name
   let col5 = col4 + 10    -- sales amount in that region

on every row
      print column col1, sr.first_name,
            column col2, sr.last_name,
            column col3, sr.region_name,
            column col4, sr.sales_amount
```

In this example, the column positions are all set relative to the column position of the previous column, with the exception of the first column which is assigned an integer constant value. This simplifies modification of the report. In order to adjust the report to allow the length of the salesperson first name to be fifteen characters instead of the current ten characters, the only modification necessary would be to the `col2` assignment statement, as follows.

```
let col2 = col1 + 15
```

This assignment adds five column positions to the starting column position for column two. Because all columns greater than column two are assigned their column position relative to column two, this will have the effect of pushing those columns over five column positions.

format SECTION—before group of/after group of CLAUSES

The INFORMIX-4GL report facility uses the `before group of/after group of` statements to provide report control breaks. The grouping logic assumes that the data is being sent to the report in a sorted fashion, and the default sequence for the grouping is the order in which they appear in the `format` section. As stated previously, this can be overridden and made specific using the `order by` clause in the `output` section of the report. If this clause is used **without** the `external` keyword, the report will ensure the report data is sorted by performing its own sort of the incoming parameters. Since the report has no knowledge of how the data is being sent to it, it will perform a complete sort.

A variety of aggregate functions can be used within the body of the report. These functions provide powerful nonprocedural capabilities which can reduce the code necessary to generate the report. They are listed in Table 5-4.

Table 5-4 Aggregate Functions

Function Name	Purpose
`sum`	Compute sum of field.
`avg`	Compute average of field.
`count`	Compute count of all rows passed.
`percent`	Compute percentage of total (count(*)/total).
`min`	Minimum value of field.
`max`	Maximum value of field.

Aggregates also support the use of a `where` clause. This clause can be used to filter the data that will be used to compute the aggregate function; only the data that fulfills the `where` clause criteria will be used in the computation of the aggregate—all other rows will be ignored. The syntax of the where clause used with aggregates is similar to that of the `where` clause used in SQL statements: a number of Boolean operators can be used to create a list of conditions which rows must meet before they are accepted. The following example demonstrates the use of the `where` clause with aggregate group functions.

```
   ...

on last row

        print  "New Jersey Sales Person Count: ", count(*) where
                                  sales_man.region = "NJ"
```

In this example, a count of rows where the salesperson's region is "NJ" is obtained on the final row of the report. The count is obtained by using the `where` clause in conjunction with the INFORMIX-4GL aggregate function `count(*)`.

BATCH PROCESSING IN REPORTS

A variety of processing can be performed within the `before group of/after group of` clauses. A common practice in production systems is to perform batch processing during the report run. This is representative of a typical batch-processing report run where database tables are updated as a report is generated. The example below demonstrates such a report.

```
{schema for acct_master and acct_balances table}

create table acct_master (        acct_code char(10),
                                  sub_code char(3)
                                  acct_group char(3),
                                  acct_name char(15) );
create table acct_balances (      acct_code char(10),
                                  acct_group char(3),
                                  acct_period smallint,
                                  acct_balance decimal(12,2) );

-----------------------------------------------------

{function and report for acct_group report }

function acct_rpt_drv()

define accts record like acct_master.sub_code,*

declare c1 cursor for
      select acct_master.acct_code,
               acct_master.acct_name,
               acct_master.acct_group,
               acct_balances.balance
```

```
      from acct_master,
            acct_balances
      where acct_master.acct_code = acct_balances.acct_code and
      acct_balances.period = 1
        order by acct_master.acct_code,
                 acct_master.acct_group

start report accts

foreach c1 into accts.*
    output to report accts( accts.* )
end foreach
finish report accts

end function
-----------------------------------------------------

report accts( act_rec )
define act_rec record like acct_master.*
define sum_amt array[2] like acct_master.balance

output
   left margin 5
   top of page "^L"

format
after group of acct_code
      let sum_amt[1] = group sum(balance)

     print "Acct Code Balance: ", sum_amt[1]

     update acct_subs
     set  sub_bal = sum_amt[1]
     where sub_code = act_rec.sub_code


after group of acct_group

      let sum_amt[2] = group sum(balance)
        print "Acct Group Balance: ", sum_amt[2]

      update acct_groups
      set    group_bal = sum_amt[2]
```

```
      where group_code = act_rec.acct_group

on last row

    print file "report.trailer"

end report
```

The goal of this report is to determine the account balances and totals for the account codes and account groups. First, the `acct_master` table is read in sorted order. On control breaks for account codes and account groups, the balance subtotals are stored in their respective tables. (The practical reason for such a *roll-up* table will be faster retrieval of rows from the subaccount table than would be possible if the accounts table had to be scanned for this information.)

The report begins by declaring a cursor for an SQL statement that will simply retrieve data from the `acct_master` table in order of account code and account group. Since the data is being retrieved in the order expected by the report, there is no need to sort the data using the report sort facility. Also, since the order in which the `after group of` statements appear in the `format` section is the order in which the groupings should be executed, there is no need to specify the order using the `order by` clause in the `format` section. If this were not the case, an explicit `order by` clause would be included in the `output` section of the report.

In the `after group of` blocks, subtotals for the balance are taken and stored in an INFORMIX-4GL variable, which is then used to update the `acct_subs` and `acct_groups` tables with the subtotals. The subtotals are taken using the built-in aggregate statements of the INFORMIX-4GL report facility: the `group sum` statements.

The statements in the `on last row` clause are triggered by the `finish report` statement. In the example above, the `on last row` clause contains a statement that uses the `print file` statement to print a report trailer. The `print file` statement will print the text document specified at the current location in the report.

`print` STATEMENT

The `print` statement makes use of the `using` clause covered in Chapter 2. It is only recognized in the `format` section of the report block. The syntax for this clause within the `print` statement is as follows.

```
print column 10, acct_name, col 15,
      acct_balance using "Balance:  $<<<<<.<<"
```

One of the more difficult aspects of printing a report is that of keeping important information together and not allowing it to be printed across pages. With INFORMIX-4GL, this can be accomplished with the `need` statement, as shown below.

```
after group of acct_sub

        need 3 lines
        print "Account Name: ",  accts.acct_name
        print "Account Group Total: ",  sum_amt[2]
        print "Sub Account Total: ",  sum_amt[1]
```

In this example, it is desirable to keep the three lines of information printed together in the same location on the report; splitting this information across pages would make it difficult to read. The `need` statement is used to keep these lines together and force a page break if necessary.

MANAGING TEXT FIELDS IN REPORTS

INFORMIX-4GL provides several useful statements for managing Blob data types within a report. In general, if the Blob data type is a Text column, the report writer has facilities for managing the data type. If the Blob data type is Binary, there are no facilities provided for managing this data type; functions, most likely C functions, would have to be called to process Binary blobs.

The Text data type is a subset of the Blob data type that provides a variable-length field limited to 2 gigabytes in size. The contents of this field are limited to the printable ASCII character set plus the control characters ^L (line-feed) and ^J (form-feed). Extensions added to the print statement in INFORMIX-4GL version 4.1 are used to format this data type. These extensions are demonstrated below.

```
format
        on every row
        print "Description : ", description_field
                        wordwrap compress right margin 80
```

In this example, the `description_field` is a Blob Text data type. It is printed using the `wordwrap` clause to wrap words at the right margin. The `compress` option is used to compress extra spaces out of the field, leaving just a single space between words. The right margin defaults to 132 bytes, but is overridden for this Blob field by using the `right margin` clause to force it to print with 80 columns. (Note that currently, the column position in the print section is ignored when the `wordwrap` option is used.)

PAGE BREAKS

The default method for forcing page breaks with INFORMIX-4GL is to print blank lines up to the length of the page. Besides being slow, this method is prone to error, and if the page length is not exact, can cause the page header to drift down the page as successive pages are printed. A preferable method of moving the printer to the next page is to use the page feed character, Control-L. This can be specified in the `output` section of the report using the following syntax.

```
output
      top of page "^L"
```

This statement will direct the INFORMIX-4GL report writer to use the Control-L character to move to the next page. Virtually all printers support using this control character to move to the next page.

A page break will trigger the `page header` section of the report block, where the code to produce the page header will be placed. In an effort to control the format of the report and ensure consistency, an `if` statement with a `print` statement used in this block must print the same number of lines in both portions of the `if/then/else` statement. The example below illustrates this behavior.

```
page header

     if rpt_rec.acct_group = 3  then
         print "Report Group Three being printed"
else
      let counter = count+1
end if
```

This statement would not be allowed. The INFORMIX-4GL report writer expects the page header to *always* be a consistent size. The belief is that if the above statement were allowed, the report would produce unbalanced formatting results. The following code would be required.

```
...

page header
     if rpt_rec.acct_group = 3  then
          print "Report Group Three being printed"
     else
          skip 1 line
          let counter=count+1
     end if
```

In this example, the report output is kept balanced by printing (skipping) one line in the `else` clause of the `if/then` statement. This will effectively compensate for the one line printed in the `then` clause above that clause.

If it is necessary to keep report information together on a page, or if an entire page is needed to print the information, the `skip to top of page` statement can be used. This will force the printer to move to the next page. The example below demonstrates the use of this statement.

```
after group of acct_group

  -- print accounts rolled into account group
  if num_accounts > 40 then -- print on a separate page
       skip to top of page
  end if
  for n = 1 to num_accts
  -- print one line for each account

                          ....
```

In this example, after the control break for subaccounts, a loop will be executed to print a list of accounts that are contained in the account group. An `if/then` statement is used to evaluate the number of accounts to be printed. If the number of accounts is greater than 40, the report will force a page break using the `skip to top of page` statement.

If the goal in printing the account detail information is to keep *all* of the account information together on one page, the above approach is flawed. The current position on the page is not taken into consideration. A better approach that would evaluate that information is shown below.

```
report sub_acct_rpt( page_length, rec )

     define page_length integer,
       rec record like ...

      ...

      after group of sub_accounts
      -- determine whether or not there is room
      -- print the list of accounts
      if ( page_length - lineno -  num_accounts ) <= 0
           then
           skip to top of page
```

```
        end if

        -- print the information
        for n = 1 to num_accts
            ...
```

In this example, the `page_length` of the report is supplied as a parameter. In addition to this parameter, the current line number on the page (`lineno`) is evaluated, along with the number of accounts to be printed. If there is not enough room to print all of the accounts together, determined by the equation `page_length - lineno - num_accounts`), then the `skip to top of page` statement is executed to move to the next page.

Modular Dynamic Reports

A more modular approach to controlling the format of a report based on parameters supplied at runtime can be accomplished by placing all relevant parameters in a record and then passing this record to the report. An example that demonstrates this approach is shown below.

```
database sales

main

define rpt_control record
      out_file char(10),
      page_length  integer,
      right_margin  integer,
      left_margin   integer
    end record

define sales_rec record like sales_person.*

let rpt_control.out_file     = "SCREEN"
let rpt_control.page_length  = 22
let rpt_control.right_margin = 80
let rpt_control.left_margin  = 3

start report sales_rpt

declare c1 cursor for
    select *
    from   sales_person
```

```
foreach c1 into sales_rec.*

   output to report sales_rpt( rpt_control.*, sales_rec.* )

end foreach

finish report sales_rpt

end main
------------------------------------------------------------

report sales_rpt( rc, sr )

define rc record
      out_file char(10),
      page_length  integer,
      right_margin  integer,
      left_margin   integer
    end record
define sr record like sales_person.*

output
    page length   55
    right margin  0
    left margin   0
    top margin    0
    bottom margin 0

format


   on every row
      print "Sales Person: ", sr.first_name, " ",sr.mi," ",
                              sr.last_name

      if lineno = rpt_control.page_length  then

          -- automatically handle output to the screen
          if rpt_control.out_file = "SCREEN"  then
             pause "Press Return to Continue"
          end if

          skip to top of page

      end if
                  ...
```

In this example, report parameters are stored in a record before calling the report. This record contains entries for page length, output file name, right margin and left margin. In the `output` section of the report, the page length is set to a number higher than the actual page length that will be enforced.

The page length parameter is evaluated as data is being printed; if the current line number on the page, contained in the built-in variable `lineno`, is greater than the specified page length, the report jumps to the start of the next page using the `skip to top of page` statement. Assuming that the `page_length` parameter entered in the control record is greater than the page length setting in the report `output` section, this effectively removes page breaks from the control of the INFORMIX-4GL report writer.

The report control record is also examined for the output device setting. If the output device is set to "SCREEN," then a `pause` statement is executed to allow the user to press return when they are ready to see the next screen displayed. This allows the report to dynamically support screen output as well as a printer device. (Since `pause` does nothing if this output is not to the screen, this conditional clause is not needed. But other code specific to screen output could be inserted here.)

REPORT ISSUES

It is not uncommon for the format of a report to be in a constant state of flux. Users often request different report formats from week to week. With some foresight and planning, report modification can be made easier, as the following examples will demonstrate.

The right margin and page length of a report can vary depending on the output data and output device. When the right margin changes, it is up to the report code to make adjustments for this change. The examples below demonstrate how to manage changes in these format parameters.

```
...
first page header

      let header1 = "Accounts Report "
      let pos = right_margin - length( header1 clipped)  / 2
      let header1 = spaces pos, header1 clipped
      let header1[1,16] = "Date: ", today
      let header1[right_margin - 12, 12 ] = "Time: ", time

      let header2= "Group Breakdown"
      let pos = right_margin - length( header2 clipped ) / 2
      let header2 = spaces pos, header2 clipped
      let header2[ right_margin - 9, 9] =  pageno using
                                              "Page: <<<"
```

```
page header
     let header2[right_margin - 9, 9] = pageno using "Page: <<<"
    print column col1, header1 clipped
    print column col1, header2 clipped
```

In this example, since the right margin can vary, it is treated as a parameter in preparing the page headers for the report. The report title is centered in a string using an equation that makes use of the current right margin to place the title in the center of the string. This equation is as follows.

```
pos = ( right_margin - length_of_string ) / 2
```

This equation identifies a starting column position that is halfway to the right margin, taking into account the length of the string to be placed at that position by padding spaces to the desired position. The next INFORMIX-4GL statement in the example places the string to be centered at that position.

```
let header1 = spaces pos, header1 clipped
```

This statement uses the built-in `spaces` function to blank-fill the string out to the starting position for the centered string. The string to be centered, in this case the contents of the string variable `header1`, are then concatenated onto this string. The page header and time are also inserted into the string using the built-in string manipulation operators of INFORMIX-4GL.

Once the two header strings have been formatted taking into account the variable right margin, they are then printed in the `page header` section of the report. Since they have been formatted in the `first page header` section executed at the start of the report, it is not necessary to format them again each time the page header is printed. However, since the page number varies, it is necessary to insert this into the string on each `page header` block iteration.

In addition to the headers, the row-by-row output of the report must take into account the varying right margin. The code below demonstrates this.

```
on every row

   let row1_data = "Account Name: ", Acct_name ...

   if right_margin > 80  then
    let row1_data = row1_data clipped, "Additional info: ",
        acct_info.group, column col3,
        acct_info.acct_status
        let row2_data = NULL
```

```
else
    let row2_data = "Additional Information: ",
                    acct_info.group, column col3,
      acct_info.acct_status
end if

   print column col1, row1_data
   if row2_data IS NOT NULL then
       print column col1, row2_data
   end if
```

In this example, the data that will be printed regardless of the length of the right margin is placed in a string. The `right_margin` variable is then evaluated to determine its length. If the length is large enough to support printing the additional data, the data is added to the `row1_data` string and the `row2_data` string is set to NULL. If the length is not long enough, the additional data is placed in the `row2_data` string. Finally, the `row1_data` string is printed and, if it is not NULL, the `row2_data` string is also printed.

MODULARITY IN REPORTS

Some general rules can be applied to enhance the modularity of reports. Whenever possible, local variables should be used within the body of the report. Since local report variables are static local scope variables, one of the most common needs for global variables in INFORMIX-4GL is eliminated. While there is often the temptation to use globals to reduce the number of values that must be passed to the report function, minimizing this practice will enhance the modularity and readability of the report.

As mentioned previously, the `first page header` section of the report is useful for initializing variables that will be used in the report. Since most INFORMIX-4GL programmers are aware that initialization can be performed in this section, it is recommended that initialization be performed in this block rather than elsewhere in the report. Local variables as well as module scope or global variables may be initialized in this block. It is also recommended that strings or variables which will be used throughout the report be initialized or formatted in this section. For instance, if a complex page header is to be printed consistently throughout the report, a series of strings could be assigned the values of the page header in the `first page header` section and then simply printed on each page of the report.

As the complexity of a report grows, it becomes more and more difficult to maintain a modular structure within the report. Since the `print` statement is only recognized within the body of a report, a function cannot be called to print the information required in a particular control block (trying to execute a `print` statement in a function would result in a compiler syntax error). Therefore, it is not uncommon for the

body of a complex report to swell to a thousand lines or more, much larger than a modular format would recommend. To reduce the number of lines of code within the report body and enhance modularity, functions can be called to load strings, which are then printed on the report. This technique is aided by the fact that the INFORMIX-4GL `let` statement allows formatting with the same `using` clause employed by the `print` statement. The following code sample demonstrates this method.

```
report accts( jrec, acctrec )
...
define line1_string char(132)


after group of acctrec.sub_accts

        let gsum = group sum(acctrec.balance)
        call sub_accts( gsum, jrec, acctrec )
                  returning line1_string

        print column 1, line1_string

....


function sub_accts( gsum, jrec, acctrec )
define ret_str1
define jrec like journals.*
define acctrec like accts_master.*
define gsum, totals decimal

select jour_name, jour_type, acct_type
into   jrec.jour_name, jrec.jour_type, acctrec.acct_type_name
from journals, accts_crx_ref, accts_master
where journals.journal_number =
      accts_crx_ref.journal_number  and
      accts_crx_ref.acct_number =
      accts_master.acct_number

let ret_str1 = gsum using "Sub-Accts total: <<<<",
               " Journal Activitity ",
               jrec.jour_name," ", jrec.total, jrec.jour_type

return ret_str1
```

```
end function
```

In this example, several records are passed into a report. Two strings are defined to the default width for INFORMIX-4GL reports. The contents of these strings is loaded with calls to the functions `sub_accts`. This function executes a series of `select` statements to retrieve data, load the data into strings, and then return those strings to the calling function.

By calling functions rather than coding the `select` statements directly in the body of the report, the report is kept more modular, and the details of the `sub_accts` function are isolated from the report program block.

SYNCHRONOUS REPORTS

Having the capability to take the same data stream and produce two distinct reports is a practical capability. INFORMIX-4GL supports such synchronous reports: sending output to two separate reports at the same time, as the following code demonstrates.

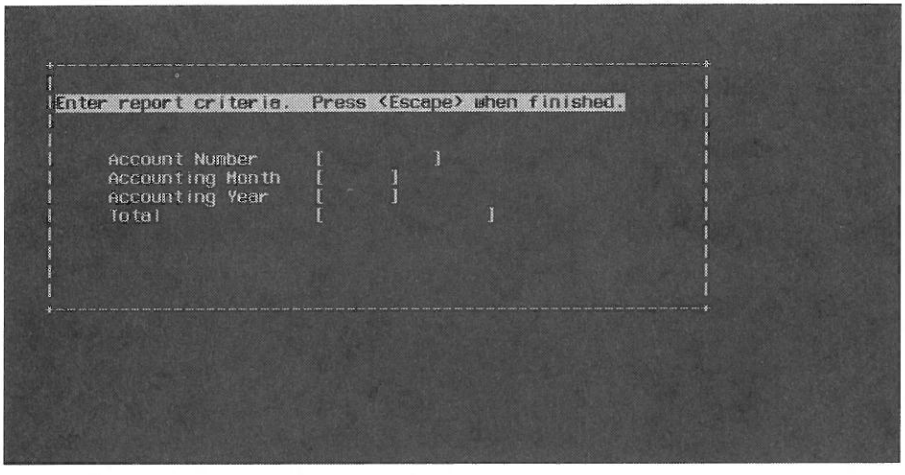
```
main

define sr record like sales_person.*

declare c1 cursor for  select *
                       from sales_person

start report sales1
start report sales2
foreach c1 into sr.*

     output to sales1( sr.* )
     output to sales2( sr.* )

end foreach

finish report sales1
finish report sales2

end main
```

In this example, two separate reports are started: `sales1` and `sales2`. Within the `foreach` loop, data is retrieved from the `sales_person` table and is then output to the two reports. At the end of the `foreach` loop, both reports are completed with the `finish report` statement.

DYNAMIC REPORTS DRIVEN BY THE `Construct` STATEMENT

A number of reports are considered dynamic in nature—their requirements change constantly. Many MIS organizations devote a great deal of programming resources to report writing and modification for these dynamic reports. However, what is often considered a *dynamic* report is really only a report with dynamic selection criteria; the format for the report remains virtually the same, only the selection criteria changes. A very useful facility for processing the varying selection criteria needed to drive these reports is query-by-example capabilities supplied by the `construct` statement (see Figure 5-2). The following example demonstrates this type of report.

Figure 5-2 Construct Statement for Report Criteria



Many organizations use a large number of bookkeeping accounts. When it is necessary to produce reports for these accounts, it is most likely that only a small subset of the accounts will be examined. For instance, if management wanted to determine expenses, they would run an accounting report for just expense accounts. Or management may want to further narrow their review to examining expenses just for January for a small set of expense accounts. Rather than write separate reports for each possible set of selection criteria, it would make more sense to simply write a single report and use different selection criteria to drive the report. Using the `construct` statement to drive this report would make this an easy task, as the example below demonstrates.

The main program block in this example merely calls two functions: the `qry_acct_totals` function, which displays the screen form and executes the `construct` statement, and the `rpt_drv` function, which receives the results of the `construct` statement, opens a cursor and then outputs data to the report.

```
database accounting

main
define where_clause char(500)

call qry_acct_totals() returning where_clause
call rpt_drv( where_clause, "acct_totals.out" )

end main
```

The `qry_acct` Function

The `qry_acct` function is used to retrieve the selection criteria for the report. It first opens a window, and then displays the screen form and a message explaining to the user how to enter the criteria. The `construct` statement is then executed. When the `construct` statement is completed, the criteria for the `where` clause of the `select` statement that will be used to drive the report will be in the `where_clause` string variable. The window is then closed, and the contents of the `where_clause` variable is returned to the calling function.

```
function qry_acct_totals()
define where_clause char(500)

open window w1 at 5,5
 with 12 rows, 60 columns
 attribute( border )

open form f1 from "acct_tot"
display form f1
message "Enter report criteria.  Press <Escape> when finished."
    attribute ( reverse )

construct by name where_clause on acct_totals.*

close window w1
return where_clause clipped

end function
```

The `rpt_drv` Function

The `rpt_drv` function receives the `where` clause criteria prepared by the previous function and the name of the output file to be produced by the report. It defines local string variables to hold the query to be used to drive the report, the record that will receive the data from the database and the name of the account being processed.

The first step taken in this function is to build the query string that will be prepared. The `where` clause criteria that was passed into the function as a string are appended to a string constant of the `select` clause of the query statement. The `select` clause must be a constant because the report that is to be called contains static report format statements that expect a fixed record structure.

Once the query statement has been created, default error checking is turned off and the query statement prepared. After the `prepare` statement, the `sqlca.sqlcode` is checked; if an error has occurred, an error message is displayed and the function returns. If the `prepare` statement has been executed successfully, a cursor is declared for the prepared statement.

If the cursor is declared successfully, the report is started. The report will be output to the file name passed into the function. A `foreach` statement is then started, which will receive the data from the database and output the data to the report. If an error is encountered, an error message will be displayed and the function will return.

```
function rpt_drv(where_clause, file_name)
define where_clause char(500),
       file_name    char(50),
       qstr         char(1000)
define at_rec record like acct_totals.*
define acct_name like accounts.acct_name

-- build the query
let qstr = "select acct_totals.*, accounts.acct_name from ",
           " acct_totals, accounts where ",
            where_clause clipped,
           " and accounts.acct_num = acct_totals.acct_num ",
           " order by acct_totals.acct_num,⇐
             acct_totals.acct_year, ",
           " acct_totals.acct_month"

whenever error continue
prepare s1 from qstr
if sqlca.sqlcode < 0 then
   call err_print( sqlca.sqlcode )
   call err_msg( "Error on prepare." )
   return
```

```
end if

declare rpt_curs cursor for s1
if sqlca.sqlcode < 0 then
   call err_print( sqlca.sqlcode )
   call err_msg( "Error on declare." )
   return
end if

start report acct_totals_rpt to file_name

-- main output loop
foreach rpt_curs into at_rec.*, acct_name

   if sqlca.sqlcode < 0 then
      call err_print( sqlca.sqlcode )
      call err_msg( "Error on fetch." )
      return
   end if

   output to report acct_totals_rpt( at_rec.*, acct_name )

end foreach

finish report acct_totals_rpt

end function
```

THE acct_totals REPORT

The `acct_totals` report receives the account totals record and the name of the current account being processed. Variables are declared to hold the column position of the columns where the report record is to be output. Variables are also declared to hold the report header strings.

In the `output` section of the report, the left margin is set to zero. This effectively removes margin control from the INFORMIX-4GL report-writing facility. Margin control will be maintained by the local variables holding the column positions for the report data.

In the `first page header` block, the variables used to print the page headers are initialized. The `all` function is used to create a string composed of a series of identical characters. In this case, a series of "=" characters is stored in the strings that will be used to print lines on the page header. These strings are loaded in the `first page`

header block to avoid the overhead of formatting these header strings each time a page header is printed.

Column positions are also loaded in the first page header block. Loading the column positions in this way will allow them to be modified easily during the application development and maintenance process. Note that all the columns are declared to be relative to the previous column (to the left of the current column). This allows a change to the second column to impact all columns that follow it sequentially on the report line.

Once the page header strings have been loaded, they are printed. INFORMIX-4GL does not execute the page header section of the report if the first page header section appears in the report, so the page header strings must be printed here. They will be printed at the column position defined in the col1 variable using a series of print statements. These same print statements will be executed again in the page header section.

A string containing the column headers is then printed. The use of the column position strings make printing these headers at the correct position an easier task. The total line is shifted to the right eight spaces to align this header with the formatting of the decimal number to be printed in this column.

```
report acct_totals_rpt( acct_rec, acct_name )
define acct_rec record like acct_totals.*
define acct_name like accounts.acct_name
define col1, col2, col3, col4  smallint
define header1, header2, header3 char(80)

output
 left margin 0

format

first page header
   let header1 = all("=", 80)
   let header2 = center( "Account Totals Report", 80 )
   let header3 = all("=", 80)


   -- set column positions
   let col1 = 4
   let col2 = col1 + 15
   let col3 = col2 + 15
   let col4 = col3 + 10

   -- print page headers
```

```
    print column col1, header1
    print column col1, header2
    print column col1, header3

    -- print column headers
    print column col1, "Acct Number",
          column col2, "Month",
          column col3, "Year",
          column (col4 + 8 ), "Total"

page header
    print header1
    print header2
    print header3
    print column col1, "Acct Number",
          column col2, "Month",
          column col3, "Year",
          column (col4 + 8), "Total"
```

In the `on every row` section of the report, the `acct_rec` record elements are printed at the column position specified in the integer variables set in the `first page header` section of the report.

In the `before group of` block for the `acct_rec.acct_num`, the name of the account to be printed is output. In the `after group of` block for the `acc_rec.acct_num` record element, the total for the account just processed is printed.

```
on every row
    print column col1, acct_rec.acct_num using "<<<<<",
          column col2, acct_rec.acct_month using "<<",
          column col3, acct_rec.acct_year  using "<<<<",
          column col4, acct_rec.acct_total using ⇐
  "###,###,###.##"


before group of acct_rec.acct_num
    skip 1 line
    print column 1, "Account Name: ", acct_name
    skip 1 line

after group of acct_rec.acct_num

    print column col4, "================"
    print column 1, "Account Total ",
          column col4, group sum( acct_rec.acct_total )
                                    using  "###,###,###.##"
    skip 1 line
```

```
end report
```

Example 5-1 Report Output for Account Totals Report

```
============================================================
                  Account Totals Report
============================================================
Acct Number      Month            Year                Total

Account Name: Electricity Exp

   1               1              1993               100.00
   1               2              1993               100.00
   1               4              1993               100.00
   1               5              1993             1,100.00
   1               6              1993             1,011.00
   1               7              1993               553.00
   1               8              1993               133.00
   1               9              1993             1,133.00
   1               10             1993               133.30
   1               11             1993               556.00
   1               12             1993                44.30
                                                 ================
Account Total                                       4,963.60


Account Name: Miscellaneous E

   11              1              1993             1,333.30
   11              2              1993               334.00
   11              3              1993             1,334.00
   11              4              1993               133.30
   11              5              1993                33.40
   11              6              1993               333.40
   11              7              1993                33.44
   11              8              1993               300.00
   11              8              1993               300.00
   11              9              1993               500.00
   11              10             1993               600.00
   11              11             1993               234.00
   11              12             1993               560.00
                                                 ================
Account Total                                       6,028.84
```

```
Account Name: Water

   2              1              1993              400.00
   2              3              1993              500.00
   2              4              1993              600.00
   2              5              1993              550.60
   2              6              1993            1,133.00
   2              7              1993              554.00
   2              8              1993              443.00
   2              9              1993              334.00
   2              10             1993            1,100.10
   2              11             1993            1,193.00
   2              12             1993            1,110.00
                                                 ==========
Account Total                                    7,917.70
```

TABLE-DRIVEN DYNAMIC REPORT

The possibilities for report formats are virtually limitless, but there are trends that tend to repeat. One such report format is a report in which a series of numbers are printed across the report page. Also, each number printed on the report is the result of a separate `select` statement, most likely one that retrieves aggregate totals. An accounting expense report which prints monthly totals across the report and account groupings on each row is an example of such a report and is shown in Example 5-2. Another example would be an accounting balance sheet where each line of the report represents a total of a series of accounts.

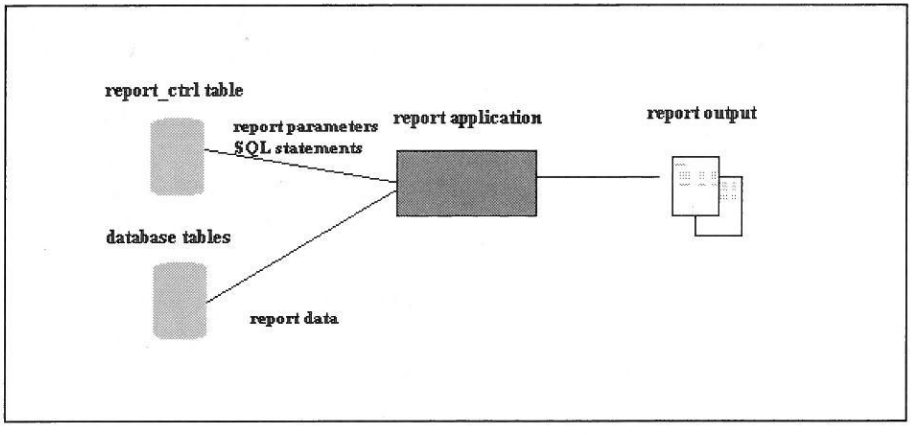
To write distinctive custom reports for each of these reporting needs could become a tedious task. A more modular approach, which would allow a significant amount of code to be reused, would be preferable. The example in Figure 5-3 below presents such an approach.

Such a dynamic report requires a table to store the information, which specifies the report format and the data to be retrieved for each column and row of the report. This is the information that will ultimately drive the report. At runtime, this table will be read. All the rows that match this report will be selected. For each row, the query statement will be prepared and executed. The result of executing the query statement will be output to the report, along with other formatting information contained in the control record. Figure 5-3 shows this process.

Example 5-2 Report Ouput for Dynamically Generated Expense Report

```
============================================================
3/19/1994              Expense Report              15:05:01
=================================================Page: 1
              Jan             Feb             Mar


Water         100.00          100.00
Phone         400.00                          500.00
Labor         600.00          100.00          123.00
Subtotals     1100.00         200.00          623.00

Cleanup       2274.40         1133.00         123.33
Misc.         1333.30         334.00          1334.00
Subtotals     3607.70         1467.00         1457.33

Grand Total   4707.70         1667.00         2080.33
```

Figure 5-3 Table Driven Dynamic Report



Each row of this table specifies the report name, information about where the data will be printed on the report, and the SQL statement used to retrieve the data. Table 5-5 shows the contents of the table used to drive this report.

```
create table report_ctrl
  (
    rept_name char(10),
    line_no smallint,
    header char(20),
    col_pos smallint,
    col_no smallint,
    sign smallint,
    action char(1),
    q_stmt char(500)
);
```

Table 5-5 Report Control Table

Column	Purpose
rept_name	The name of the report.
line_no	The line number of the report. This relates to the rows of the report.
header	If line_no = 0, this is the header or report title for the report. If line_no is not zero, this is the header or title for the row.
col_pos	The column position where the data to be retrieved will be printed.
col_no	The column number or row number for this row. This number is used to identify the data in this column. It will be used to store subtotals and grand totals and to sort the data retrieved from the report_ctrl table.
sign	Stores the sign (+/-) of the data in this row-column. If the data in this column to be printed is negative, a negative 1 is stored here. If the data to be printed is positive, a positive 1 is stored here.
action	A single-byte flag for this row. Indicates the action to be performed. Possible settings are listed below.
q_stmt	Stores the query statement that will be used to retrieve the data for this row, column.

The `main` Program Block and the `rpt_drv` Function

The `main` program block simply calls the `rpt_drv` function with the name of the report, the column width of the report, and the file where the report will be output, as shown below.

The `rpt_drv` function will read rows from the `report_ctrl` table that describes the report and then pass the contents of each row to a function which will process the row. First, a cursor is declared to read all rows from the `report_ctrl` table that match the report name being run. Once the cursor is declared, the report is started and output to the file name that has been passed into the function.

The `foreach` loop is then executed, and all elements of the `report_ctrl` table are retrieved into the `report_ctrl` record. The `action` element of the `report_ctrl` record is a flag that indicates the type of row being processed. The possible settings for the action flag are shown in Table 5-6.

Table 5-6 Action Flag Settings

Flag Setting	Purpose
N	This indicates that the data in the row-column is numeric data. This means that the `proc_line` function must be called to process the data.
S	This indicates that the row-column will be used to trigger a subtotal.
G	This indicates the row-column will be used to trigger a grand total.
H	This is a header row. The contents of the `header` column will be printed as the report header.

If the action flag is set to "N," the `report_ctrl` record is passed to the `proc_line` function, which will process the query contained in the record and return an amount into a decimal variable. The amount variable, the report control record and the column width for the report are then output to the report. When the `foreach` loop is finished, the report is closed with the `finish report` statement.

```
database testit
main

call rpt_drv( "Accounting", 80, "acctg.out" )

end main


function rpt_drv( rept_name, col_width, file_name )
define rept_name like report_ctrl.rept_name
```

```
define col_width smallint,
       file_name char(50)
define rc record like report_ctrl.*
define amt decimal


declare c1 cursor for
  select *
  from   report_ctrl ct
  where  ct.rept_name = rept_name
  order by line_no

start report sql_rpt to file_name

foreach c1 into rc.*

    if rc.action = "N" then
       call proc_line( rc.* ) returning amt
    end if
    output to report sql_rpt( rc.*, amt, col_width )

end foreach

finish report sql_rpt

end function
```

The `proc_line` Function

The `proc_line` function takes the report control record as a parameter. A decimal is defined to be used as a return value. The default error checking mode is turned off, and the statement contained in the `q_stmt` element of the report control record is prepared. If an error has been encountered, an error message is displayed along with the query statement that failed. A cursor is then declared for the prepared query statement. The cursor is opened, and the data is fetched and returned to the calling function.

Note that there is no attempt to read additional rows from the cursor; it is asserted that the query will only retrieve a *single* row of numeric data, presumably an aggregate sum of a series of numeric values.

```
function proc_line( rc )
define rc record like report_ctrl.*
define retval decimal

whenever error continue
```

```
prepare rpt_stmt from rc.q_stmt
if sqlca.sqlcode < 0 then
  error "Error on prepare.", sqlca.sqlcode
  initialize retval.* to NULL
  return retval.*
end if

declare rpt_curs cursor for rpt_stmt

open rpt_curs

fetch rpt_curs into retval

return retval * rc.sign

end function
```

The `sql_rpt` Report

The `sql_rpt` report receives as parameters the report control record, a decimal value of the amount for the row, and the column width of the report. Variables are declared to hold the header of the report and the current line of the report. Because of the manner in which data is processed in this report, subtotals and grand totals cannot be processed by report aggregates (group sum, group subtotals, etc.) (This would not make any sense, since report aggregates only act on parameters passed into the report; they would not recognize the multiple columns output on the report.) Therefore, two arrays of 50 dimensions each are declared to hold the totals and grand totals for the report. Variables are also declared to be used as counter variables and flags within the report.

In the output section of the report, the left margin is set to zero to allow the report complete control of the placement of the report columns. In the `first page header` section, initializations are performed. The subtotal and grand total flags are set to false, and the subtotal and grand total arrays initialized to zero; the logic of the report demands that this be done.

Since the report header must be built before other rows are output from the report, the report control data must be input in such a way that the report header row from the control table will be the first row passed into the report. That has been accomplished in this example by setting the column number (the `col_no` column from report_ctrl table) for the header record to zero. Since the report control records are sorted in ascending column number and all other nonheader rows have column numbers greater than zero, this will cause the report control header record to be passed into the report first. The code in the `first page header` section takes the additional precaution of verifying that

the `action` element of the `rc` record is set to "H." In fact, if the first row passed to the report is the header record, there is no need to check this value.

Within the `first page header` control block, the report header is created and stored in a series of three strings. The `all` function is used to initialize strings that will be used to store the column totals. Since these strings will be inserted into substrings of the current line string variable, it is necessary to set the entire string to blanks so that indeterminate values are not left in portions of the string which have not been explicitly assigned.

The title of the report is contained in the `header` element of the report control record. Also, the column headers are contained in the `q_stmt` element of the report control record; these are merely assigned to the `colheader` string. The date, time and page number are also placed in the report header strings. All header information is placed relative to the column width parameter passed into the function.

Once they have been created, the header strings and the column header string are output. The only information in the third page header string that changes is the page number. This is reinserted into the `header3` string on each page.

```
report sql_rpt( rc, amt, col_width )
define rc record like report_ctrl.*
define amt decimal,
       col_width  smallint
define header1, header2, header3, colheader char(132)
define tstr char(50)
define curr_line char(132)
define n,x,colno smallint
define sub_totals array[50] of decimal
define grand_totals array[50] of decimal
define sub_total_flag, grand_total_flag smallint

output
   left margin 0

format

first page header

-- initialize
let sub_total_flag   = FALSE
let grand_total_flag = FALSE

for n = 1 to 50
    let sub_totals[n]    = 0
```

```
    let grand_totals[n] = 0
end for

-- these need to be passed in 1st to force initializations
if rc.action = "H"  then -- build the header

   let curr_line = all(" ", col_width )
   let colheader = all(" ", col_width )
   let header1   = all("=", col_width )
   let header2   = all(" ", col_width )
   let header2 = center( rc.header, col_width)

   let colheader = rc.q_stmt     this field contains column⇐
     headers
   let tstr      = today
   let n         = length(tstr clipped)
   let header2[1,n+1] = tstr clipped
   let tstr = time
   let n = length( tstr clipped)
   let header2[ col_width - n, col_width ] = tstr clipped
   let header3 = all("=", col_width )

end if

   let tstr = pageno using "Page: <<<"
   let n = length( tstr clipped )
   let header3[ col_width - n , col_width ] = tstr

   print header1   clipped
   print header2   clipped
   print header3   clipped
   print colheader clipped
   skip 1 line


Page header

   let tstr = pageno using "Page: <<<"
   let n = length( tstr clipped )
   let header3[ col_width - n , col_width ] = tstr

    print header1 clipped
    print header2 clipped
```

```
print header3 clipped
print colheader clipped
skip 1 line
```

THE on every row REPORT BLOCK

The `on every row` section of the report performs a great deal of the processing. The report flags used in this report are "N" for numeric output, "S" for subtotals and "G" for grand totals. If a row has any of these flags set, then the amount supplied must be stored at the correct offset in the current line (the `curr_line` variable) string. The code currently implemented allows 10 characters for the amount column. The offsets for loading the amount into the current line string are calculated and stored in local variables. This is largely necessary due to peculiarities in the INFORMIX-4GL parser, which occasionally complains about expressions in a substring reference. The amount will therefore be inserted into the current line strings into the substring of the starting column position passed in the report control record plus 10 bytes.

First, based on the setting of the action flag in the report control record element, an amount will be stored in the current line substring. If the action flag is "S," a subtotal should be inserted into the current line for the column number being processed. Columns are numbered on the report in a left-to-right fashion, with the first column being one and the maximum column being 50. The column number is used as an array index into the subtotal and grand total arrays, and since the arrays are defined as 50 element arrays, they should only accept element values between one and 50.

If the `action` record element is set to "S," "G" or "N", then the amount value from the appropriate array is placed in the current line string at the column position (the report control `col_pos` record element) offset. A series of `if/then` statements is used to determine which variable to use. If the action flag is "S" or "G," then the subtotal flag or grand total flag is set to TRUE. This flag will be used to trigger other actions later in the program. The row header is also loaded into the current line at the substring, starting at the first column position and extending for the length of the header string element of the report control record.

Next, the amount value is tested. If the amount is NULL, it is set to zero to allow sane mathematical processing later in the program. The subtotal flag is then evaluated to determine whether or not the subtotal flags should be cleared. If the subtotal flag is set and the current row being processed is not a subtotal row (`rc.action != S`), the subtotals can be cleared. (Since grand totals are only printed once, there is no reason to clear them.) The subtotal flag is then set to false so that subsequent iterations will not execute this code unnecessarily.

If the action flag for the current row is set to "N," this is a numeric line and needs to be added to both the subtotal accumulator and the grand total accumulator for that column.

Since the current line is made up of a series of rows from the report control table, each of the rows have the same report line number. Each row of the report control table should not be output as it is passed to the report. It is only when *all* of the report control rows for the current report line number have been passed to the report that the current line (contained in the current line string) should be output to the report output file. After the line is output, the current line string is reinitialized. On the final row of the report, if the grand total flag is set, then grand totals are output to the report file.

```
on every row

if rc.action = "N" or rc.action = "S"  or rc.action = "G" then
   let n = rc.col_pos
   let x = n + 10
   let colno = rc.col_no

    -- load the amount
   if rc.action = "S" then  -- subtotal
       let curr_line[n,x] = sub_totals[colno]
       let sub_total_flag = TRUE
   end if

   if rc.action = "G" then -- grand total
      let curr_line[n,x] = grand_totals[colno]
      let grand_total_flag = TRUE
   end if

   if rc.action = "N" then
      let curr_line[n, x] = amt
   end if

   -- load the row header
   let n = 1
   let x = n + length( rc.header clipped )
   let curr_line[n,x] = rc.header clipped

   -- process the amount or subtotal
   if rc.action = "N" then
      if amt IS NULL then
         let amt = 0
      end if
```

```
   end if

end if

   -- clear subtotals if necessary
if sub_total_flag and rc.action != "S"  then -- clear the⇐
     subtotals

      for n = 1 to 50
         let sub_totals[n] = 0
      end for

      skip 1 line

      let sub_total_flag = FALSE

end if

   if rc.action = "N"  then
      let sub_totals[colno]    = sub_totals[colno] + amt  -- ⇐
    accumulate the total
      let grand_totals[colno] = grand_totals[colno] + amt  -- ⇐
    accumulate the grand total
   end if


after group of rc.line_no
   print curr_line clipped
   let   curr_line = all( " ", col_width)


on last row

  if grand_total_flag then
     skip 1 line
  end if
  print curr_line clipped

end report
```

Chapter 6

Functions in INFORMIX-4GL

One of the most significant features of INFORMIX-4GL is the ability to create functions. This feature alone makes the language a standout among 4-GLs. Not only does Informix- 4GL allow the creation of functions, but it does so in a manner that is modular and block-oriented in the tradition of such third-generation languages such as Pascal, Algol and C.

4GL functions have a modular design; functions have a block beginning designated by the `function function_name` statement, and a block ending designated by the `end function` statement. Within this block, any variables declared in the variable declaration section of the function have a local scope; they are only recognized within the block scope of that function. (Note that a local variable name declaration masks the declaration of any variables declared with the same name with module or global scope.)

This treatment of variable name scope mirrors that of C or Pascal and is a very useful language design feature. Programmers are free to name all array index variables without fear of altering other array index variables in use elsewhere in the program. Additionally, the use of local variables makes more efficient use of memory because space for local variables is allocated on function entry and deallocated on return from the function.

Unfortunately, the use of local variables does not come easily to programmers more familiar with languages like COBOL, in which variables are declared in a single statement block and then used throughout the program; a style of programming antithetical to that of modular programming.

INFORMIX-4GL does not force applications to be written in a modular manner. However, the benefits of limiting variable scope, especially with local variables in functions, are significant. The benefits of modular programming in INFORMIX-4GL should not be underestimated.

Function Parameters

In INFORMIX-4GL, functions can receive parameters of any data type except arrays, and can return parameters of any data type except arrays. Also, unlike many languages, INFORMIX-4GL can return multiple parameters.

Functions in INFORMIX-4GL are *call-by-value*. A parameter passed to a function is passed as a *value*; it cannot be modified by that function. There is currently no capability to pass a function parameter *by reference*.

The data type of incoming parameters must be defined within the variable declaration section of the function body. These data types should correspond to the data types, in ordinal position, of the parameters used to call the function, but this is not required. The reason for this is INFORMIX-4GL will, when possible, convert the data type passed to the function to the appropriate data type for that parameter. Failure to define the data types of incoming function parameters will result in a compiler error.

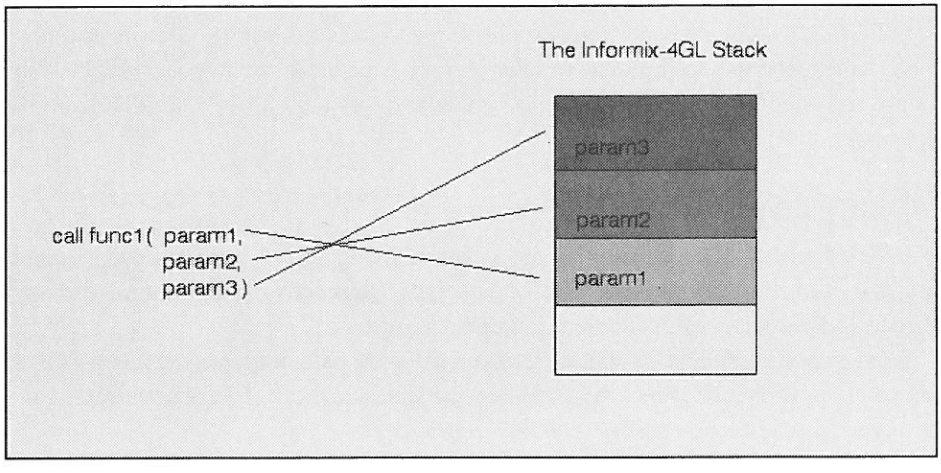
IMPLEMENTATION

The implementation of functions in INFORMIX-4GL is similar to that of other modular languages. Function parameters are pushed onto the INFORMIX-4GL stack in a left-to-right order. As they are pushed onto the stack, the data type of the variable is recorded on the stack. When INFORMIX-4GL calls the function, the function is sent the number of parameters pushed onto the stack. This stack is shown in Figure 6-1.

Before the function retrieves its parameters from the stack, it will first retrieve the number of parameters it has been passed. If this number is not the correct number of parameters, the function will signal a fatal error and the program will abort.

If the correct number of parameters has been passed, they are retrieved from the stack in reverse order of the order in which they were placed on the stack, (i.e. right to left.) As the parameters are retrieved from the stack, the data type required for the parameter is compared to the data type of the parameter being retrieved. If the parameter can be converted, it will be; if not, a fatal error is returned and program execution will be terminated. INFORMIX-4GL does not explicitly support passing a variable number of parameters to a function—the number of parameters is effectively fixed.

Figure 6-1 The INFORMIX-4GL Stack



Once all parameters are retrieved, the body of the function is executed. Any valid INFORMIX-4GL statement may be placed within the body of a function except a function declaration. On finishing execution of the function block, INFORMIX-4GL will place return parameters on the stack, and then return, or send, the number of parameters placed on the stack. Return parameters are treated as are the function parameters: they are retrieved in reverse order and data type conversion will take place if it is sensible. On return from the function, all space used for storage of local variables is deallocated—the variables cease to exist.

All parameter type conversions and testing of the number of parameters are not performed until runtime; the INFORMIX-4GL compiler will not check the validity of function calls..

Functions as Expressions

Function calls can also be treated as expressions in INFORMIX-4GL. The following statements are valid.

```
--the function isvalid_state() returns integer 0 for
--false and 1 for true

let x = isvalid_state("CA")


define x integer
if x then
  call do_something()
end if
```

In the above code, the assignment statement

```
let x = isvalid_state("CA")
```

automatically assigns the return value of `isvalid_state` to the variable `x`. This assumes that the function `isvalid_state` returns only one parameter (if it did not, a runtime error would occur).

INFORMIX-4GL Boolean expressions also allow functions as expressions. The following code is valid.

```
if  isvalid_state("CA")  then
      call do_something()
end if
```

Note how making use of this capability reduces the number of lines of code required to make the validation and creates more modular, readable code. More complex, nested function expressions are also allowed, as shown below.

```
let state = "ca"
let country = "us"

if isvalid_state( upshift(state) )       and
    isvalid_country( upshift(country))   and
    isvalid_zip("08833")                  then
    call do_something()
end if
```

USE OF VARIABLES WITHIN A FUNCTION

As mentioned previously, INFORMIX-4GL variables declared within a function have local scope—they are only visible within that function. The space for these variables is allocated on entry to the function and deallocated on exit from the function. This means that between successive function calls, the variable does *not* retain its value. In most cases, this may not be a problem. However, in cases where the function must remember something about a previous call, a variable that can retain its value is needed. Since the use of global variables is discouraged in modular coding, something short of a global variable is desired.

The solution is to use a variable declaration with module scope. This is a variable that is only visible to the INFORMIX-4GL source code module (source code file) in which it is declared. These variables are simply declared outside of a function block and outside of a global statement block, as shown below.

```
--
-- declare module scope variables
--
define first_pass_flag smallint
define pass_counter    integer

--
-- function declarations start here
--

function get_data(rec_key)
#
# Purpose: to retrieve data from the tab1 table
#
define rec_key  like tab1.key_col
```

```
define rec_var  record like tab1.*

--
-- assert first_pass variable is initialized to TRUE in main
-- We only want to execute the following code once, on the
-- first pass through the function
--
if first_pass then

    declare c1 cursor for
      select *
      from tab1
      where key_col = rec_key

    open c1

    let first_pass = false

end if

fetch c1 into rec_var.*

--
-- assert pass_counter was initialized to 0 in main
let pass_counter = pass_counter + 1

call display_counter( pass_counter )

return rec_var.*

end function
```

In the above example, the declaration of the two module scope variables, `first_pass` and `pass_counter`, allow the function `get_data` to *remember* whether or not it has been called before. If it has been called before, it will avoid the overhead of having to declare and then open the cursor (an expensive database operation). Also, a count of the number of passes can be kept in the variable `pass_counter`, which is passed to a function that presumably would display a count of the number of records returned so far. Note that `first_pass` and `pass_counter` would be visible to other functions in the source code module and could be modified by them, but they would not be visible or modifiable in other source code modules.

PASSING A VARIABLE NUMBER OF PARAMETERS TO FUNCTIONS

INFORMIX-4GL does not explicitly allow a variable number of parameters to be passed to a function. However, by making use of some simple string manipulation capabilities of INFORMIX-4GL, a variable number of parameters can be passed to, and returned from, an INFORMIX-4GL function.

The key to this capability is to place delimited function parameters into a string and then pass that string as a function parameter. This method is shown below.

```
let table_list = "table1, table2, table3 "
let column_list = "col1, col2, col3 "
call get_data( table_list, column_list )
```

The function would then take the delimited string and parse the parameters, placing each parameter into a variable. These variables would then be used by the function as function parameters. The string manipulation required to do this is shown below.

```
function get_data( tables, cols )
define tables char(100)
define cols   char(100)
define x,y    smallint
define len    smallint
define table_parms array[10] of char(20)
define col_parms   array[10] of char(20)

--
-- initialize the parameter strings.
--
for n = 1 to 10
    for y = 1 to 20
        let table_parms[n][y] = " " -- a blank character
   end for
end for

--
-- move through tables string, getting parameters
--
let len = length( tables )
let n = 1
let y = 1

for x = 1 to len
  --
```

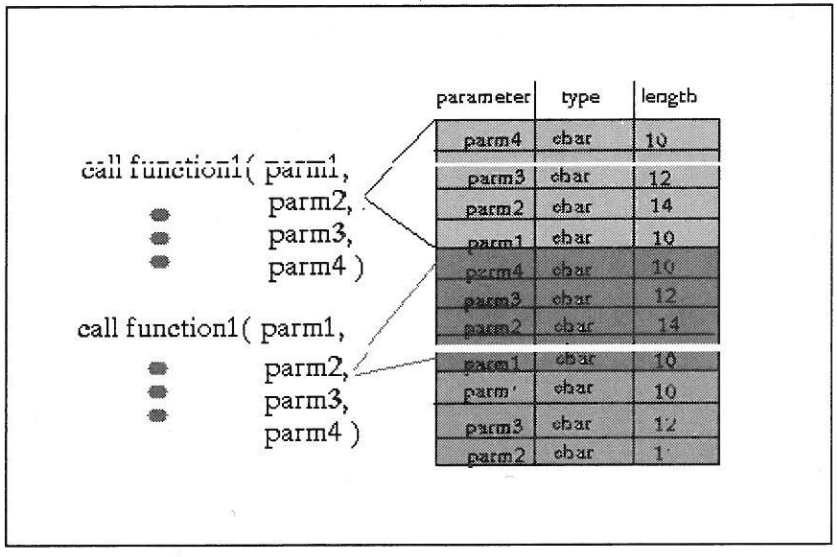
```
    -- 'copy' to parameter array
    --

    let table_parms[y][n] = tables[x]
    let n = n + 1

    --
    -- test to see if we are on a delimiter.  If so,
    -- jump to the next parameter
    --

    if tables[x] = "," then -- terminate the parameter
       let x = x + 1        -- move past the delimiter
       let y = y + 1        -- next parameter
       let n = 1            -- start at the beginning
    end if

end for

--
-- do the same for the columns list passed
-- into the function
--
...
```

In the above example, the input parameters are parsed in a `for` loop that manages to get all the parameters passed up to a safe limit (supplied by the built-in `length` function) and store them in an array of character variables. However, before the character array can be used, it must be initialized to a set of safe values. This is necessary for two reasons. First, as mentioned previously, declared variables in INFORMIX-4GL always have an indeterminate value (the p-code version runner makes this appear otherwise, but a programmer should always code as though the value were indeterminate). Second, character strings in INFORMIX-4GL are always assumed to be padded to their defined length. INFORMIX-4GL assignment statements that reference an entire string will pad the string with blanks if the whole string is being referenced. However, in cases where individual elements of the string are being assigned, the string is not (and cannot) be padded with blanks.

The `initialize` statement is usually used to initialize these variables, but in the case of character variables, initialize will simply set the first character to a NULL character. (This behavior improves performance—instead of having to modify 20 bytes of a 20-byte string, only one byte need be modified.)

For our purposes in the code example above, this won't work since we are directly addressing elements of the string with assignment statements. If we did not initialize all elements of the string to spaces, odd characters would creep into the end of the string following the last byte we explicitly assigned.

Figure 6-2 The INFORMIX-4GL Stack



Once the array of strings has been initialized in the example, a `for` loop is used to parse the parameters from the string passed to the function. Various integers used to serve as indexes are initialized to appropriate values.

A `for` loop is then started for the entire length of the string containing the parameters. For each iteration of the `for` loop, the contents of the string containing the parameters is copied, byte by byte, to the array of string parameters. When a delimiter character is detected, in this case a comma, the index variables that keep track of the current parameter array element are incremented and reset. The result is that at the end of the `for` loop, the `tables_parms` array contains all of the tables' parameters passed into the function.

Note that this example does not provide coding to handle instances where more than 10 parameters are passed into the function. A more complete example of using this method of passing function parameters to create a generic validation function is shown later in this chapter.

RECURSION IN INFORMIX-4GL

One of the more powerful function-related language capabilities is available in INFORMIX-4GL: recursion. Recursion is generally considered one of the necessities of a modular third-generation language. INFORMIX-4GL supports recursion in both its procedural code and nonprocedural code with only minor limitations. INFORMIX-4GL places no restriction on the levels of recursion allowed: recursion can continue as long as there is memory available to support the associated stack operations.

Recursion essentially allows a function to call itself. Since INFORMIX-4GL is using a stack and placing arguments on the stack in a push-down fashion, function parameters will not become confused by recursion. However, recursion comes with a cost: function calls require stack operations, and stack operations can be expensive in terms of resource usage. Often, even though a recursive solution is available, an iterative programming solution is chosen to avoid the overhead of recursion.

Recursion offers a relatively simple, elegant solution to problems that would be difficult to express with an iterative algorithm. The following code provides an example of recursion in INFORMIX-4GL.

```
--
--define module scope variable to hold counter value
--
define depth smallint -- indicates the depth of recursion

main

let depth = 0
call recurse()

end main

function recurse()
define col smallint

--
-- increment level counter
--
let levels = levels + 1

--
-- stop at a depth of 20
--
if levels = 21  then
   display "Finished run." at 8,10
```

```
    sleep 2
    return
end if

--
-- display message
--
display levels using "Recurse level << "
                at 10,10
let col = 9 + levels
display "*" at 12, col


--
-- go even deeper
--
call recurse()

end function
```

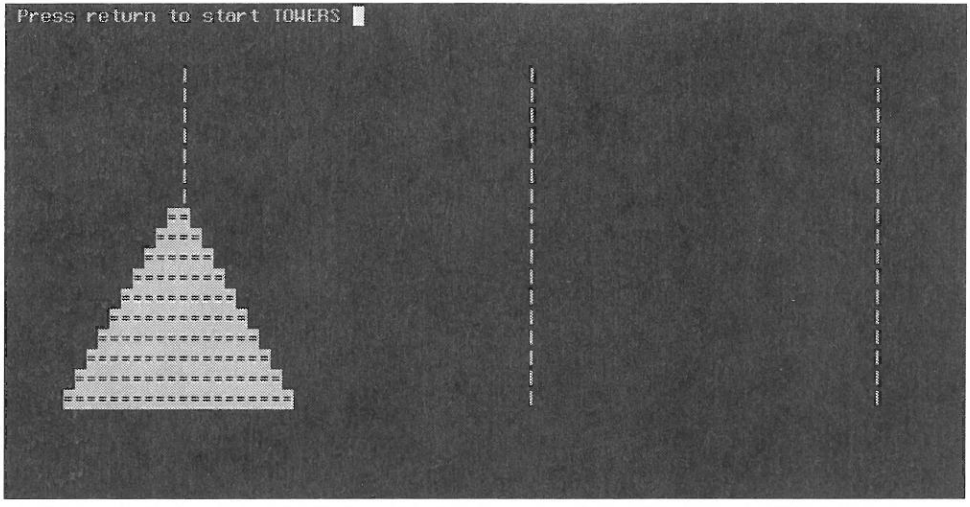
In the example above, a module scope variable is declared to provide a counter variable that will hold its value between calls. On each call, the counter is incremented and the recursion levels count displayed along with a text graph that indicates the same information. When 20 levels of recursion have been reached (on the 21st iteration), the program will exit.

TOWERS OF HANOI PROBLEM

The story is told of the priests of the Temple of Brahma attempting to solve a problem using disks and a series of towers. The problem involved a set of disks of varying size, from a large diameter to progressively smaller diameters. All of the disks had to be moved from a start disk, where they were all initially stacked, to a final disk. The disks were required to be placed on the initial peg ranging in size from the smallest disk on top to the largest disk on the bottom of the peg. When the disks were moved, they could only be placed one at a time on either an empty peg or a disk of a larger size.

The monks believed that if the problem could be solved, the end of the world would be imminent. While the monks worked with 64 disks, the problem has been reduced to fewer disks and solved using recursion with 3GL programming languages. Fortunately, this has not precipitated the end of the world, though the expense of recursion has been known to wreak havoc on some smaller computer systems.

Figure 6-3 The Towers of Hanoi Application



Because of its recursive capabilities, INFORMIX-4GL can be applied to the Towers of Hanoi problem. The following code demonstrates an INFORMIX-4GL solution to the Towers of Hanoi.

The solution to this problem can be determined empirically using a small number of disks. If three disks are used, with the top disk being disk number one and the bottom one being disk number three, then the solution is:

move disk 1 from peg 1 to peg 3
move disk 2 from peg 1 to peg 2
move disk 1 from peg 3 to peg 2
move disk 3 from peg 1 to peg 3
move disk 1 from peg 2 to peg 1
move disk 2 from peg 2 to peg 3
move disk 1 from peg 1 to peg 3

Since the solution for three disks can be reduced to this series of moves, the solution for an unlimited number of disks on three pegs could be made by isolating the solution for three disks and then recursively calling that procedure until all the disks had been moved. The calls would be as follows:

```
call towers( num_of_disks, start_peg, third_peg, final_peg )
```

This function must be called for each *level* until all levels have been traversed. Calls to the function would therefore be as follows:

```
  if disk_num = 1 then                    -- final move
    call move_disk( disk_num, final_peg )
  else
    call towers( disk_num - 1, start_peg, final_peg, third_peg )
    call move_disk( disk_num, final_peg )
    call towers( disk_num - 1, third_peg, start_peg, final_peg )
  end if
```

This algorithm would push the program to the deepest level (disk_num=1) before any disks would be moved. Therefore, the problem will essentially be solved from the bottom up, in recursive terms.

Once the algorithm has been created, the next task in writing the application is that of creating data structures to track the movements of the disks and the contents of the pegs. Given these data structures, a scheme can then be developed for displaying the movement of the disks to the screen.

The movement of the disks will be tracked by disk rather than through the pegs. The disk data structure will contain the size of the disk, an image of the disk, the current peg location and the screen coordinates of the disk. The peg data structure will contain the row and column of the peg and will be used to keep track of the current top row of the peg (this is the row where new disks will be placed). Since these data structures must maintain their value between calls, they will be declared with module scope.

```
--
-- the data structure used to track the movement of the disks
--
define disk array[64] of record
        image  char(30),              -- image of disk
        disk_size smallint,           -- size of disk
        peg_location smallint,        -- location of peg
        disk_row     smallint,        -- current disk row
        disk_col     smallint         -- current disk col
  end record

--
-- the data structure used to track the state of the pegs
--
define peg array[10] of record
  peg_col     smallint, -peg column. will not change
  top_row     smallint, -top row. will not change. top of peg
  curr_row    smallint, -current row
  bottom_row  smallint  -bott row.changes based on disk placement
 end record
```

```
define tower_bottom,
       tower_top,
       spacing,
       sleep_val smallint
define funct_calls integer  -- count the number of towers() calls

--------------------------------------------------------------------

main
define disk_num,
       start_peg,
       third_peg,
       final_peg,
       n          smallint
define num_disks, num_pegs smallint
define start_dt, end_dt datetime hour to second

--
-- initialize variables
--
let tower_bottom = 20    -- bottom row of towers
let tower_top    = 3     -- top of tower
let spacing      = 30    -- spacing between towers
let funct_calls  = 0     -- calls to towers()


--
--get command line args and call towers function with those
--parameters
--
if num_args() > 0  then
   let num_disks = arg_val( 1 )
end if

if num_args() > 1  then
   let sleep_val = arg_val( 2 )
else
   let sleep_val = 0
end if

let start_peg = 1
let third_peg = 2
let final_peg = 3
```

```
let num_pegs  = 3

--
--a 24 row by 80 column screen has problems with over 14 disks
--so limit to 14 disks
--
if num_disks > 14  then
   let num_disks = 14
end if

if num_disks = 0 then
   let num_disks = 10     -- the default
end if

--
-- create the data structures to track disk movement
--
call build_objs( num_disks, num_pegs )

--
-- display the screen
--
call draw_pegs( num_pegs )
call draw_disks( num_disks )

prompt "Press return to start TOWERS " for n

let start_dt = current

--
-- recursively call towers till game is complete
--
call towers( num_disks, start_peg, third_peg, final_peg )

let  end_dt   = current

display "Processed  ", funct_calls using "<<<<<<< calls.", " Start
time: ",
         start_dt, ". End time: ", end_dt clipped

prompt "Press return to end " for n

end main
```

First, the two primary data structures for the program are declared with module scope since they must retain their value between function calls. Next, a number of other variables that are used throughout the program are declared. These variables will be used to track screen positions, parameters passed into the program, and a counter of the number of function calls made to the `towers` function.

In the main program block, several variables are declared to be used as parameters to the towers function. Next, the module scope variables that establish the screen location for the towers and disks are set. The value of these variables are examined in the `draw_pegs` and `build_objs` functions. The command line arguments are examined, and program parameters are set depending on the existence and/or content of the command line arguments. (Note that a parameter for `sleep_val` is provided. The contents of this variable are used to determine whether or not to pause between screen display operations.)

The `build_objs` function, which will be examined later, will *build* the data structures for the objects to be displayed on the screen: the disks and the pegs. The `draw_pegs` and `draw_disks` functions are then used to display these objects to the screen.

Once the screen has been displayed, the user is then prompted to press return to begin the program. Once the return key is pressed, the start time is recorded in a datetime variable, and the `towers` function is called with the parameters that have been set previously in the main program block. Upon completion of the numerous recursive calls this function will make, the end time is recorded in a datetime variable. Finally, some statistics from the current run are displayed on the terminal screen, and a prompt is provided to allow the user to read the statistics before the program finishes and clears the screen.

```
function towers ( disk_num, start_peg, third_peg,
final_peg )
define disk_num, start_peg, third_peg, final_peg smallint

if disk_num = 1 then
  call move_disk( disk_num, final_peg )
else
  call towers( disk_num - 1, start_peg, final_peg,
third_peg )
  call move_disk( disk_num, final_peg )
  call towers( disk_num - 1, third_peg, start_peg,
final_peg )
end if

let funct_calls = funct_calls + 1

end function
```

The towers function provides an explicit description of the actions to be performed if the disk_num passed is one. For any other value, the else condition of the if/then clause will be executed. The else condition will have the disk_num decremented and passed to the towers function recursively. When this happens, the current state of the function stack will be placed on the program stack. This *function state* will include the current program location and the value of the variables being used in the function.

As the code in this function block indicates, recursive calls will be made to the function call immediately after the else clause. These calls will continue to be made until the disk_num variable has a value of one. Only when this level has been reached will the calls to move_disk begin to be made. Once the move_disk call has been made in the else clause, recursive calls are made at the same program stack level to the towers function. Once again, these calls will continue to be made until the disk number is equal to one. At that level the move_disk function will be called, and the program will once again begin to traverse up the stack, executing the else condition of the if/then statement until the previous level has been reached. Before returning to the previous level, a counter variable is incremented to track the number of calls made to the towers function.

```
function move_disk( disk_num, peg_num )
--
-- move the disk disk_num to peg peg_num.
-- This is implemented by deleting the
-- disk image at the current disk position and then
-- displaying the disk image at the new disk location.
--
define disk_num, peg_num smallint

if sleep_val then              -- sleep if sleep_val is set
   sleep sleep_val
end if

--
-- pass the disk_num and the current peg for that disk
--
call delete_disk( disk_num, disk[ disk_num ].peg_location )

--
-- store the new peg number
--
let disk[ disk_num ].peg_location = peg_num
```

```
--
-- pass the disk_num and the new peg for that disk
--
call place_disk( disk_num, disk[ disk_num ].peg_location )

end function

-----------------------------------------------------

function delete_disk( disk_num, peg_num )
--
-- delete the disk disk_num from its current location
--
define disk_num, peg_num smallint
define dimage char(30)
define col_pos smallint
--
-- create an image of blanks to clear the disk image from the
-- screen
--
let dimage = all(" ", disk[disk_num].disk_size)

--
-- delete the old image

--
let col_pos = peg[peg_num].peg_col-(disk[disk_num].disk_size / 2)
display dimage
       at disk[disk_num].disk_row, col_pos

--
-- display the peg character
--
display "|" at disk[disk_num].disk_row, disk[disk_num].disk_col

--
-- move the bottom row up
--
if ( peg[peg_num].bottom_row != tower_bottom ) then
   let peg[peg_num].bottom_row = peg[peg_num].bottom_row + 1
end if
```

```
end function

------------------------------------------------------------
function place_disk( disk_num, peg_num )
--
-- place the disk on peg peg_num.  Determine the new
-- location of the disk based on the bottom row coordinates of
-- the peg and the column location of the peg.
--
define disk_num, peg_num, col_pos smallint


--
-- set the new coordinates for the disk
--
let disk[disk_num].disk_col = peg[peg_num].peg_col
let disk[disk_num].disk_row = peg[peg_num].bottom_row

--
-- center the disk on the peg
--
let col_pos = peg[peg_num].peg_col - (disk[disk_num].disk_size/2)

--
-- display the disk in reverse video
--
display disk[disk_num].image clipped
                at disk[disk_num].disk_row,  col_pos
          attribute( reverse )

--
-- decrement the bottom row of the peg
-- (move it up one row )
--
let peg[peg_num].bottom_row = peg[peg_num].bottom_row - 1

end function
--
```

The next set of functions is used to display the disks being manipulated by the `towers` function. In order to simulate the movement of the disks, the old image of the disk at the old disk location is first deleted, and then the new image at the new location of

the disk is displayed. The top-level call to the functions that perform these actions is the `move_disk` function. This function will pause (or sleep) for the number of seconds defined in the main program block. The number of seconds to sleep are passed to the program as a parameter, and, if not supplied by the user, default to zero seconds for no sleep interval.

The `delete_disk` function is passed the disk number to delete and the peg number at which the disk currently resides. A number of spaces the size of the disk image are placed in a string. This string of spaces is then displayed at the current location of the disk in order to clear the disk from the screen. The peg character is then redisplayed and then, if the location of the disk is *not* the bottom row of the peg, the bottom row of the peg is decremented. This will cause any subsequent attempts to display a disk on this peg to display it at that row, the row from which the previous disk has been deleted. Note that the column position of the disk is based on the expression

```
let col_pos = peg[peg_num].peg_col-(disk[disk_num].disk_size / 2)
```

Since half the disk size is used to offset the disk from the column position of the peg, this has the effect of centering the disk on the peg.

The `place_disk` function is passed the disk number of the disk being moved and the peg number of the peg to which the disk is being moved. This function will set the disk row and disk column positions as the peg bottom row and peg column, respectively. The new location of the disk is stored in the record being used to track the location of the disk. Once the disk has been moved and displayed, the bottom row of the peg is moved upwards.

```
function build_objs( num_disks, num_pegs )
--
-- build/initialize the data objects used in the program
--
define num_disks, num_pegs smallint
define disk_size, n smallint

--
-- define peg coordinates
--
for n = 1 to num_pegs

   if n = 1 then
      let peg[n].peg_col = spacing / 2
   else
      let peg[n].peg_col = spacing + peg[n - 1].peg_col
   end if
```

```
    let peg[n].top_row      = tower_top
    let peg[n].bottom_row   = tower_bottom

end for

--
-- build the peg structure
-- start on first peg
--
for n = num_disks to 1 step -1

  let disk[n].peg_location = 1
  let disk[n].disk_size    = n * 2
  let disk[n].image        = all( "=", disk[n].disk_size )
  let disk[n].disk_col     = peg[1].peg_col
  let disk[n].disk_row     = peg[1].bottom_row
  let peg[1].bottom_row    = peg[1].bottom_row - 1

end for

end function
------------------------------------------------------------
function all( pchar, num )
--
-- return the string of size num composed of character
pchar
--
define pchar   char(1)
define num     smallint
define ret_str char(100)
define n       smallint

for n = 1 to num
   let ret_str[n] = pchar
end for

return ret_str[1,n] clipped

end function
```

The `build_objs` function is used to create the objects to be displayed on the screen. It is passed the number of disks to be created and the number of pegs being used.

First, a `for` loop is used to initialize the data structures for the pegs. The size of disk image and the spacing of the pegs is scaled based on the number of disks and pegs.

The next `for` loop in the code is used to initialize the disk structures. Note that the loop starts from the highest disk number and decrements the counter. This will make the smallest disk disk number one, as is needed by the application.

By game rules, all towers must start by being placed together on the first peg, so the peg location parameter is set as a constant in this function. The disk size is created based on the number of the disk multiplied by the number two. The image of the disk is created using a string of characters supplied by the `all` function. The disk column and row are then set based on the column and row of the peg. The bottom row of the peg is then decremented, allowing the top of the peg to move upwards.

A factor that limits running this program with more than 14 disks is the 24-rows-by-80-columns terminal screen. These terminals would not support the disk images created by the algorithm used in the `build_objs` function.

The `all` function emulates a function supplied by many early versions of BASIC. It accepts a character parameter and an integer number. It then creates a local copy of a string of the characters passed by using a `for` loop, which will iterate the loop for the number of characters that has been requested. The string is then returned to the calling function.

```
function draw_pegs( num_pegs )
--
-- draw the pegs on the terminal screen
--
define num_pegs, n,x, cur_row, cur_col smallint

for n = 1 to num_pegs

  let cur_row = peg[n].top_row + 1

  for x = 1 to peg[n].bottom_row - peg[n].top_row
      display "|" at cur_row, peg[n].peg_col
      let cur_row = cur_row + 1
  end for

end for

end function
```

```
--------------------------
function draw_disks( number_disks )
--
-- draw the disks on the screen
--
define number_disks smallint
define n,x smallint

for n = 1 to number_disks

   --
   -- center the disks
   --
   let x = peg[1].peg_col - (disk[n].disk_size/2)
   display disk[n].image clipped at disk[n].disk_row, x
                  attribute ( reverse )
end for

end function
```

Since the data structures provided carry most of the information needed to be displayed on the screen, drawing the disks is simply a matter of iterating through these data structures and displaying the contents on the screen.

The `draw_pegs` function is passed the number of pegs that have been created. The function executes a loop for the number of pegs created and then displays the pegs on the screen.

The `draw_disks` function iterates through the disk structures/arrays and displays the disks on the screen. Note that the disk size is once again divided by two in order to center the disk on the peg.

An Example of Recursion — The Parts Explosion Problem

The use of recursion is not limited to such esoteric problems as the Towers of Hanoi; recursion provides solutions to some common relational database problems. The most notable of these problems is that of the self-referencing table. A very good *real-world* example of this is shown in the parts explosion problem.

The table definition below is for an auto parts database. Parts stored in the table are made up of component parts that are themselves parts. By definition, the level of self-referencing is unlimited. A good implementation of recursion will support recursion to a depth limited only by available memory, which is the case with INFORMIX-4GL.

The Informix-SQL script shown below creates the parts database, creates the parts table and indices, and then loads data into the parts database.

```
{ create the parts database }
create database partsdb;

{ create the parts table and indexes }
create table parts ( pname             char(20),
                     pnum              integer,
                     component_of_pnum integer );
create index pidx1 on parts ( pnum );
create index pidx2 on parts ( component_of_pnum );

{ load the data for the parts table }
insert into parts values ( "alternator-p1", 1, 0 );
insert into parts values ( "gear-ax1z", 2, 1 );
insert into parts values ( "bearing", 3, 1 );
insert into parts values ( "stabilizer-rod", 4, 2 );
insert into parts values ( "coil-assembly", 5, 1 );
insert into parts values ( "wire-assembly", 6, 5 );
insert into parts values ( "wire-frame", 7, 5 );
insert into parts values ( "coil-assembly-gasket", 8, 5 );
insert into parts values ( "clips-a2", 9, 6 );
insert into parts values ( "washer-w2", 10, 5 );
insert into parts values ( "clip-w2", 11, 7 );
```

The following is the INFORMIX-4GL programmatic solution to the parts explosion problem. This code uses the data above and recurses to a depth of three levels to provide a listing of all auto parts, their component parts, and the component parts of those component parts.

```
database partsdb

--
-- define module scope variables to use as program
-- counters and program constants
--
define levels     smallint
define MAX_LEVELS smallint

main
define pr record like parts.*

declare c1 cursor for
select  *
from    parts
```

```
    --
    -- Initialization
    --
    let MAX_LEVELS = 5
    let levels     = 0

    foreach c1 into pr.*

      display pr.pnum using " Part number <<<<  Name: ",
                            pr.pname

      call get_parts( pr.pnum )

    end foreach

end main

--------------------------------------------------------
function create_cursor( plevel, lpnum )
--
-- DECLARE and OPEN the cursor needed for
-- this level number
-- hardcoded limit of five levels.
--
define plevel smallint
define lpnum  like parts.pnum
define qstr char(100)

let qstr = " select * from parts where ",
           " parts.component_of_pnum = ? "

case plevel
    when 1  prepare stmt1 from qstr
            declare c11 cursor for stmt1
            open c11 using lpnum

    when 2  prepare stmt2 from qstr
            declare c12 cursor for stmt2
            open c12 using lpnum

    when 3  prepare stmt3 from qstr
            declare c13 cursor for stmt3
            open c13 using lpnum
```

```
      when 4  prepare stmt4 from qstr
              declare c14 cursor for stmt4
              open c14 using lpnum

      when 5  prepare stmt5 from qstr
              declare c15 cursor for stmt5
              open c15 using lpnum

end case

end function

---------------------------------
function fetch_cursor( plevel )
define plevel smallint
define pr record like parts.*

            case plevel
                when 1
                     fetch c11 into pr.*
                when 2
                     fetch c12 into pr.*
                when 3
                     fetch c13 into pr.*
                when 4
                     fetch c14 into pr.*
                when 5
                    fetch c15 into pr.*
        end case

return pr.*

end function

---------------------------------

function get_parts( lpnum )
--
-- recursively retrieve component parts
--
define lpnum  like parts.pnum     -- part master
define cpr record like parts.*    -- components of master
```

```
let levels = levels + 1

if levels > MAX_LEVELS then
                display  "Too many levels"
                exit program
end if

call create_cursor( levels, lpnum )

call fetch_cursor( levels ) returning cpr.*

--
-- Continue to loop as long as data is found
-- at this level
--
while cpr.pnum IS NOT NULL  and sqlca.sqlcode = 0
      display cpr.pnum using
            "    Component part number: <<<<   Name: ",
              cpr.pname clipped, lpnum using
                      " Component of <<<<",
                      levels using " Recurse level: <<"
            --
            -- Now get its component parts, if any
            --
            call get_parts( cpr.pnum )

            call fetch_cursor( levels ) returning cpr.*
end while
let levels = levels - 1  -- going back one level

end function
```

This program begins by declaring a cursor to read all rows from the `parts` table. The cursor is then opened implicitly by the `foreach` statement, which will continue to read data from this cursor until no more rows are found.

For every row read, the function `get_parts` is called. The `get_parts` function will determine whether or not component parts exist for this part and will display information on those component parts if they exist.

INFORMIX-4GL supports recursion completely, but unfortunately, the cursors it creates do not. Cursors created in INFORMIX-4GL have module scope, not local scope. Their data space is allocated from the heap, and they retain their value between

successive function calls. (This was an intentional design — cursors are expected to retain their values as long as they are open.)

This behavior causes problems in a recursive function. If a single cursor were opened in the `get_parts` function, and `get_parts` was called recursively in a `foreach` loop, the `get_parts` function would terminate the `foreach` loop before returning to the `get_parts` function one level up. On return to the `get_parts` function, the program would begin executing within the `foreach` loop, but the cursor controlling the `foreach` loop will have been closed by the previous recursive call to `get_parts` — it will no longer be opened.

The solution to this problem is to write code that will create a series of cursors to potentially be used. This is done in the `create_cursor` function that is passed the parts number parameter and the level parameter. Based on the recursion level, a cursor is created which selects all component parts for that part number. The cursor is also opened in this function.

Instead of a `foreach` loop, a `while` loop is used as a control structure for retrieving the rows for the component parts. The `fetch_rows` function is called with the recursion level as a parameter. Since the recursion level was used as an *index* value when we created the cursor, we can also use it to fetch rows from the same cursor.

As each row is retrieved, the part number, part name, master part number (the part number of which this part is a component), and recursion level are displayed. Before the end of the `while` loop, the fetch_cursor function is called again to retrieve any additional rows. If no rows are found, the `while` loop is terminated. On exit from the `while` loop, the `levels` variable is decremented because program execution is returning to the previous level.

This implementation of the parts explosion problem is limited to five levels of recursion, but because of the modular nature of this code, it would be fairly simple to add additional levels to the `create_cursor` and `fetch_cursor` functions and adjust the MAX_LEVELS value accordingly.

Generic Functions Using the Variable Parameters Example

Using the parameter-passing scheme shown in this chapter, a programmer can develop generic functions that can be reused to provide the same functionality for a variety of purposes. The following example shows how generic functions can be applied to the problem of validations in a relational database.

In a master-detail relationship, whenever a detail record is added, its foreign key should have a corresponding master record in the master table. If not, an error has occurred, and the program should indicate to the user that the record is invalid. Rather than continually rewriting code to execute a query for each existing master-detail

relationship, it would be easier to simply write a single generic function to validate any given master-detail relationship. The function could be called as follows:

```
if not validate("customers", "cust_number", "123435") then
      error "invalid customer number. Please re-enter."
            Next field cust_number
end if
```

Over the development cycle of an application, such a function could reduce the lines of code needed to produce an application. However, to implement this function, the `prepare` statement must be used to compose and execute the query, which is not known until the program is run. The code for such a validation function is listed below.

```
   function validate     ( ptable_name,
                           pfield_name,
                           pvalue)

   define ptable_name char(25)    -- table to search
   define pfield_name char(25)    -- field to compare
   define pvalue      char(25)    -- value for where clause
   define qstr        char(255)   -- string to hold query
   define cnt         smallint    -- stores the count(*)
   define ret_value smallint      -- returns true or false

  let qstr = " select count(*)", " from ",
              ptable_name  clipped,
             " where ",  pfield_name clipped,
             " = ", "'", pvalue clipped,"'"

prepare qstr_exec
from qstr

declare qstr_curs cursor for qstr_exec
open qstr_curs

fetch qstr_curs into cnt

if cnt = 0  then
   return FALSE
else
   return FALSE
end if

end function
```

The validate function shown above accepts three parameters: the name of the table for which the validation will be performed, the field name that will be examined and the value that must be validated. The query used to perform the validation is created by simply concatenating the various portions of the query, which have been passed as parameters to the function, together into a string variable. Since the function is only concerned with the existence of a record, there is no need to select values from the tables used in the query; the built-in SQL aggregate function `count(*)` is adequate.

Once the SQL statement has been pieced together in the string variable, the variable is in turn prepared into a statement identifier. Since the statement is returning a value, it must declare a cursor for the value; this is done on the next line.

The cursor is then opened, and the return value from the query is then fetched into the variable `cnt`. If the value of `cnt` is zero, it can be assumed that no records exist for the value passed, and the constant value FALSE is returned. If the value of `cnt` is nonzero, the constant value TRUE is returned.

Issues in Creating Generic Functions

This particular example lacked error checking. In using the `prepare` statement, the syntax of the statement cannot be checked until runtime, since the statement does not exist at compile time. Additionally, other information about the query, such as the existence of database objects (tables, columns), their data types, and the validity of any expressions in the SQL statement, are not checked until runtime. A more complete function would process and return errors to some generalized error-handling routine.

There can also be problems with the data types being passed into a generic function; they are unknown at compile time. INFORMIX-4GL does not include any statement that explicitly converts data types, but it does include implicit data type conversion based on an internal set of conversion rules. These conversion rules allow any data type to be converted into a character string, with the only possible problem being one of truncation (which returns a warning with string data types, not an error).

The example above makes use of this conversion rule by defining the value parameter in the function as a character string of sufficient width to handle most values. Also, note that when the Informix-SQL statement is concatenated, the value of the variable `pValue` will be enclosed in single quotes in the resulting Informix-SQL statement. The Informix-SQL parser will interpret this value as a string, and if the left-hand value in the expression is numeric, it will automatically convert the value into a numeric (a `decimal(13)`) in order to evaluate the expression. If the left-hand value is a character string, no conversion is required, and the evaluation is that of two-character strings.

If the quotes had been omitted and the left-hand value were a character string such as 'John Doe,' the SQL parser would quickly become confused, since the expression would then appear in the `where` clause as

```
WHERE customer_name = John Doe
```

The parser would not know how to process the 'Doe' part of the expression and would return the ubiquitous -201 - SQL syntax error.

Additionally, if the left-hand part of the expression were a character string being used to store a number (not uncommon in a key field), and the right-hand value was a number but was not enclosed in quotes, the parser would see the expression

```
WHERE customer_numb = 1234567
```

The parser would interpret the right-hand side of the expression as an integer, and the evaluation, while not producing an error, may not return the results expected.

Passing a Variable Number of Function Parameters in INFORMIX-4GL

Some validations may require that a concatenated key be referenced. This means that more than one field parameter and more than one value parameter would have to be passed. In order to make the validation function more generic, it must be able to process these varying numbers of function parameters. The code fragment below shows an example of a validation function receiving additional parameters.

```
if validate_table("stock",
                  "stock_num,manu_code",
                  "101, hro")            then
        error "this stock number/manufacturer ",
              " code is already on file.",
              " Please re-enter."
        next field stock_num
end if
```

This function is similar to the first validation function, since it accepts arguments for the table name, field name and value to be tested. Where it differs is that it optionally allows additional parameters to be passed in each string function parameter using a comma as a delimiter. This allows a variable number of function parameters to be passed in each parameter string.

The bulk of the code in this function is used to process the variable number of arguments passed and then create the `where` clause portion of the Informix-SQL statement. Once this portion has been created, the remaining code is very similar to the previous validate function. The sample code for the new validation function is shown and explained below.

```
function  validate_table(pTable_name,
                         pField_name,
                         pValue)
```

```
define pTable_name char(25)     -- table name to query
define pField_name char(100)    -- field name to query
define pValue      char(100)    -- query value
define qstr        char(255)    -- string to hold query
define cnt         integer      -- integer to hold count
define x,y,i,z,n,
       val_len, fld_len   smallint
define pFld, pVal
         array[5] of char(25) -- arrays to store varargs
define where_part char(500)    -- string to hold
                               -- where clause
define tmpstr     char(50)     -- temporary string

let cnt = countchar(",", pField_name)

--
-- First, initialize the array elements,
-- else NULL characters creep in
for n = 1 to 5
    for x = 1 to 25
        let pVal[n][x] = " "
        let pFld[n][x] = " "
    end for
end for

if cnt > 0 then
  let fld_len = length(pField_name)
  let val_len = length(pValue)
  let y = 1
  let i = 1
  let cnt = cnt + 1 -- need this many elements

  for n = 1 to cnt

     let x = 1

     -- Process the fields
     while pField_name[y] != "," and
          y < (fld_len + 1) and
          pField_name[y] IS NOT NULL

          if pValue[i] IS NOT NULL then
             let pFld[n][x] = pField_name[y]
```

```
            let x = x + 1
         end if

         let y = y + 1
    end while
    let y = y + 1                -- move past the ","

    --
    -- Process the corresponding values
    --
   let z = 1
   while pValue[i] != ","  and
         i < (val_len + 1) and
         pValue[i] IS  NOT NULL

        if pValue[i] IS NOT NULL then
           let pVal[n][z] = pValue[i]
           let z = z + 1
        end if

        let i = i + 1

    end while

   let i = i + 1                 -- move past the ","

 end for

 --
 -- Create the WHERE clause
 --
 let where_part = " where "
 for n = 1 to cnt

     let where_part = where_part clipped," ",
                      pFld[n] clipped,
                      " = '", pVal[n] clipped, "'"
     if n != cnt  then
          let where_part = where_part clipped,  " and "
     end if
  end for

else  -- Only one column  passed
```

```
      let where_part = " where ",
                       pField_name clipped,
                       " = ",     "'",
                       pValue clipped, "'"
end if

whenever error continue

let qstr = " select count(*) ",
           " from ",
            pTable_name clipped,
            where_part clipped

prepare qstr_exec from qstr
declare qstr_curs cursor for qstr_exec
open    qstr_curs
fetch   qstr_curs into cnt

--
-- Simply return FALSE if error occurred
--
if sqlca.sqlcode < 0   then
   let cnt = 0
end if

whenever error stop

if cnt = 0  then
     return FALSE
else
     return TRUE
end if

end function


----------------------------------------------------
function countchar(pchar, sstr)
-- Purpose: to find the occurrences of pchar
-- in the string sstr
-- returns the number of occurrences found
--
define pchar char(1)
```

```
define sstr  char(75)
define cnt,n   smallint

let n   = 1
let cnt = 0
while n < length(sstr clipped)
     if sstr[n] = pchar then
        let cnt = cnt + 1
     end if

    let n = n + 1

end while

return cnt

end function
```

In the `validate_table` function above, the `countchar` function is first called to determine the number of comma delimiter characters contained in the field list parameter. The function uses this count to determine whether or not the additional processing is necessary to handle a variable number of arguments. If only one parameter is passed for the table and columns parameters, all of the additional processing performed for multiple column and table parameters can be avoided, thus allowing the function to perform more efficiently.

Next, the string variables used to hold the individual table and column parameters are explicitly initialized as they were in the previous example. They are then tested for whether or not variable parameters were passed into the function. If so, the function must determine the length of the strings containing the variable number of arguments. The counter variables are then initialized, and a loop is executed for the number of individual parameters contained in the function argument. It is assumed that there are a corresponding number of value parameters for each field name parameter passed, so both parameters are processed in the same loop.

In the first `while` loop, for each parameter passed, a loop is used to examine the field name parameter string and copy all characters between the comma separators into the array of character strings used to hold the field name parameters. The second `while` loop performs the same operation for the corresponding value parameters. At the end of both loops, a `let` statement forces the string position counter to skip over the comma delimiter character.

Once both the array of fields and corresponding values have been loaded into the array, the `where` clause is created. Since all values are now contained in arrays, this

operation is simple and can be handled with a single `for` loop. Note that once again, values are enclosed in single quotes so that the Informix-SQL parser will initially evaluate them as strings.

Once the `where` clause has been created, the query string is created by concatenating the various portions of the query together in a single string. A statement id is prepared, a cursor is declared for the statement id, and the cursor is opened and then fetched into a value, which is then evaluated before returning TRUE or FALSE.

This function effectively turns off error checking using the `whenever error continue` statement before executing the `prepare`, `declare`, `open` and `fetch` statements. It then evaluates the `sqlca.sqlcode` for an error condition and merely returns FALSE if an error occurred, leaving it for the programmer to figure out why incorrect results are returned during the debug process. This function could be modified to return specific values if an error occurred, but this may preclude using the function in a Boolean expression, which merely evaluates TRUE or FALSE conditions.

ADDING ADDITIONAL FUNCTIONALITY WITH GENERIC FUNCTIONS

Another version of the `validate` function is shown below. In this version, a `where` clause with query criteria is passed to the function. This allows for more flexibility in using the function, but requires a different set of parameters to be passed. Instead of calling the `validate` function as done in the previous example:

```
let table_name = "customer"
let cust_numb  = "1234567"

if not validate( table_name,
                 "cust_number",
                  cust_numb )  then
   error "Invalid customer number. Please re-enter."
   next field cust_number
end if
```

the function would be called as follows:

```
let table_name = "customer"
let str = "cust_number = ", "1234567"

if not validate( table_name, "cust_number", str )  then
    error "invalid customer number. Please re-enter."
    next field cust_number

end if
```

The following code example shows how this version of the `validate` function would be written.

```
function validate ( ptable_name, where_clause )
define ptable_name      char(150)
define where_clause     char(300)

define qry_string  char(500)
define cnt         integer

let cnt = 0

let qry_string = "select count(*) from ", ptable_name clipped,
       " where ", where_clause clipped

prepare prp_id from qry_string
declare qry_curs cursor for prp_id

open qry_curs

fetch qry_curs into cnt

close qry_curs

free prp_id

return cnt
end function
```

In this example, the `where` clause is passed into the function as a second parameter. This parameter is simply concatenated onto an SQL `select` statement, which will retrieve the aggregate value `count(*)` for the parameters passed. This query statement is then prepared and executed, and the integer value returned from the `count(*)` value aggregate function is returned. Since a zero value is a Boolean false, and a non-zero value a Boolean true in INFORMIX-4GL, this function could be used in an `if/then` clause in the same manner as in the previous example. If `count(*)` is greater than zero, the function will return a non-zero value and test true in a Boolean statement. If `count(*)` is zero, the function will return zero, which will test false in a Boolean statement.

SUMMARY

Functions represent one of the more powerful features of INFORMIX-4GL. Using functions, a programmer can effectively extend the language to perform additional tasks as needed. When used as part of a modular programming effort, a library of functions can be created and then shared by a group of programmers.

When functions are used together to create other functions, programmers effectively reuse code and reduce the number of lines of code required to develop an application. This methodology can shorten the application development process and make maintaining an application easier since program operations are isolated and grouped in an understandable set of modular function libraries. Using functions as part of modular programming effort is the cornerstone of the coding style demonstrated in this book. The use of modular, generic functions will be demonstrated in a variety of other examples throughout this text.

Chapter 7

Input Array and Display Array: Pick Lists and Zoom Functions

INFORMIX-4GL provides a number of basic, nonprocedural screen interaction statements. Chapter 5 introduced these screen interaction statements. This chapter covers variations on two of these statements: the `input array` and `display array` statements. These statements act much as do their simple screen form counterparts, `input` and `display`, and even share many of the same clauses. Their major difference is that they provide interaction with a program array and a scrolling region on the screen.

When combined with the complex logic possible with procedural statements, complete, full-featured applications can be developed. Several examples of such applications are demonstrated in this chapter.

While the language does provide some significant nonprocedural screen interaction statements, it does not provide a facility for a table *zoom* or a *pick list*. A pick list is little more than an `input array` or `display array` statement executed within a window. A zoom function utilizes SQL statement processing which retrieves a set of columns from a table or tables and then displays those columns in a pick list. With very little effort, generalized routines can be written to perform these functions. Several such routines are presented in this chapter.

DISPLAY ARRAY EXAMPLES: THE ZOOM FUNCTIONS

The following functions present a high-level access to relatively *low-level* functionality of the `display array` statement. Given parameters for the name of a table, one or more columns in that table, `where` clause criteria and labels to display for the corresponding columns, a *zoom* screen is displayed. This zoom screen allows the user to move the cursor to any particular row, scrolling the region on the screen if necessary, and ultimately to choose an item from the list. Upon exiting the `display array` statement, the value chosen is stored and then returned to the function that called the zoom function.

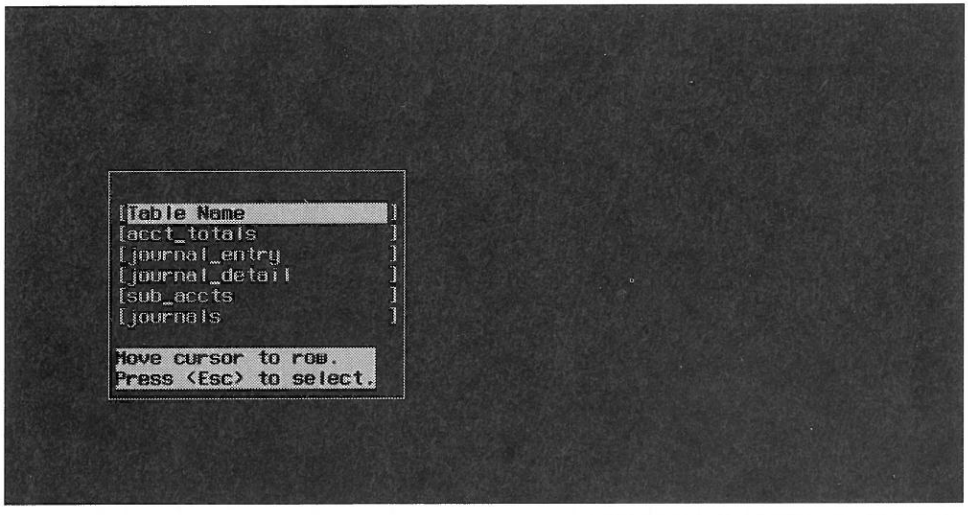
The `zoom1` Function

The `zoom1` function will receive parameters for the table or table names, the single column, `where` clause criteria and a label to use for the column to be displayed. It will return the item the user selected. The sample screen output for this function is shown in Figure 7-1.

If the `zoom1` function were being called to display a list of tables that were not system tables, as shown above, then it would be called as follows.

```
call zoom1( "systables", "tabname", "tabid > 99", "Table Name" )
     returning ltabname
```

Figure 7-1 The zoom1 Function



The screen form for the `zoom1` function is presented above. Note that the database specified is the `formonly` database, thus allowing the form to be used with any database. This generality is needed if the function is to be used as a generic function.

The first screen field is given the label `l1`. The remaining screen fields are all given the label `f1` and, in the instructions section of the form, are identified as part of the screen record `zoom1_rec`. The screen record is needed in order for the form is to be used with the `display array` statement.

```
database formonly
screen
{
[l1                    ]
[f1                    ]
[f1                    ]
[f1                    ]
[f1                    ]
[f1                    ]
}
attributes
l1 = formonly.label1;
```

```
f1 = formonly.field1;
instructions

screen record zoom1_rec[5] ( field1 )
```

Implementation of the zoom1 Function

The `zoom1` function first defines the table and column `where` clause and column label parameters and then defines an array of 100 elements that will be used to hold the data. A variable is used to hold the array limit of 100; this will make the code more readable and allow the array size to be changed in the future by simply changing the dimensions of the array in the `define` statement and changing the number assigned to the `pArr_Limit` variable in the `let` statement. A variable is also declared to hold the query string that will be created.

The query clause is constructed using the column name that has been passed. The `where` clause is then evaluated. If it is NOT NULL, the `where` clause string is appended to the string being used to build the query.

Next, error checking is turned off, the query string prepared and the cursor opened. If there is an error during the cursor open, an error message is displayed and the function returns a NULL value. Note that there is no error checking after the `prepare` statement, the assertion being that the programmer has passed the correct parameters and it is a cursor open that would fail. Also note that there is no `database` statement issued to open the database; it is presumed that the database is open when this function is called.

The next few lines of the function are used to retrieve the data from the database. A `foreach` loop retrieves the rows from the database and loads them into the array that will be used to display the values to the screen. A counter is incremented during each loop, and if the counter is beyond the limits of the screen array, an error message is displayed and the `foreach` loop is exited.

The counter variable is evaluated after the loop that loads the data; if it is zero, no data has been loaded, and the function will display an error message indicating that no rows have been found and return.

If data is found, a window will be opened and the column label displayed to the screen. The `set_count` function is then called. When used in conjunction with the `display array` statement, the `set_count` function has the effect of creating a limit to the `display array`.

After calling `set_count`, a message with instructions for the user is displayed on the terminal screen and the `display array` statement is executed. Upon exit of the `display array` statement, the `arr_curr` function is called to get the row in which the cursor is currently resting. The window is then closed, and the cursor returns the value of the current `display array` row when the ACCEPT key was pressed.

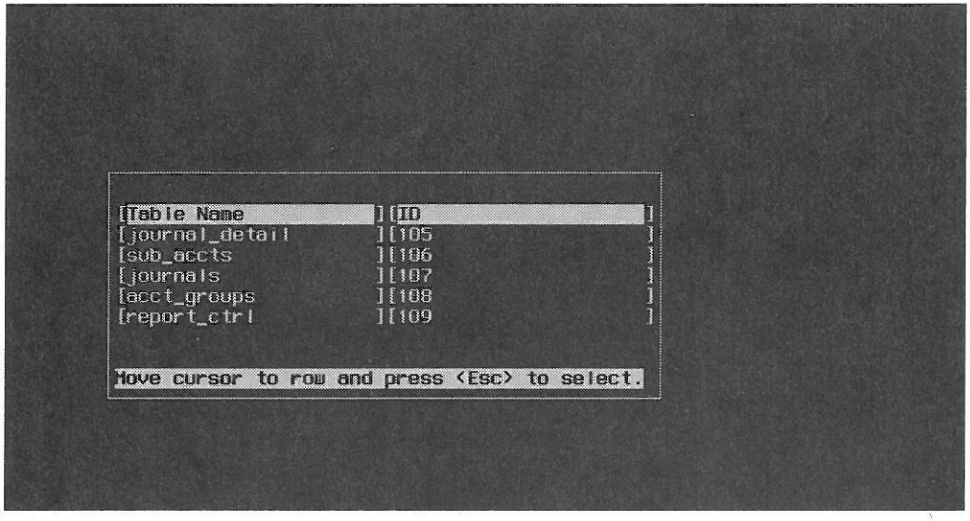
```
function zoom1(pTable, pCol1, pWhere, pLabel1 )
define pTable  char(40)
define pCol1   char(40)
define pLabel1 char(20)

define pDis_arr array[100] of
   record col1 char(20)
  end record

define pWhere char(100)
define pcount smallint

define Arr_Limit smallint
define qstr char(1000)

let Arr_Limit = 100

-- create select statement
let qstr = "select ",pCol1 clipped

let qstr = qstr clipped, " from ", pTable clipped

if pWhere IS NOT NULL then
    let qstr = qstr clipped," where ", pWhere clipped
end if

whenever error continue

prepare _curs1 from qstr
open _curs1

if sqlca.sqlcode < 0   or
   sqlca.sqlcode = 100  then
   error sqlca.sqlcode using "Error: <<<< on open. "
   sleep 1
   return ""
end if


--
-- retrieve the data
-- load the array list
--
```

```
let pcount = 1
foreach _curs1 into pDis_arr[pcount].col1

    if pcount + 1 > Arr_Limit then
       error "Too many rows."
       exit foreach
    end if
    let pcount = pcount + 1
end foreach
let pcount = pcount - 1

if pcount = 0  then

   error "No rows found."

   return  ""

end if

open window _wif1 at 10,10
     with form "zoom1"
     attribute ( border,
                 message line last,
                 form line first )

display pLabel1
   to   label1
 attribute (reverse)

call set_count(pcount)

display "Move cursor to row.    " at 9,1 attribute( reverse )
message "Press <Esc> to select."
     attribute( reverse )

display array pDis_arr to zoom1_rec.*

let pcount = arr_curr()

close window _wif1
return  pDis_arr[pcount].col1 -- return the user's choice

end function
```

The zoom2 Function

The zoom2 function is similar to the zoom1 function, except that two fields are displayed on the screen instead of one. The functionality and flow of control are virtually the same (see Figure 7-2.)

Figure 7-2 The zoom2 Function



The screen form for the zoom2 function creates two field labels: l1 and l2. These will be used to hold the field names for the fields displayed in the array. The screen record array is then composed of an array of the fields f1 and f2.

```
database formonly
screen
{

[l1                    ][l2                    ]

[f1                    ][f2                    ]
[f1                    ][f2                    ]
[f1                    ][f2                    ]
[f1                    ][f2                    ]
[f1                    ][f2                    ]

}
attributes
l1 = formonly.label1;
l2 = formonly.label2;
```

```
  f1 = formonly.field1;
  f2 = formonly.field2;

  instructions

  screen record zoom2_rec[5] ( field1, field2 )
```

A sample program implementing the zoom2 function is shown below. The function is called with parameters for the table or tables from which data will be retrieved, and one parameter for each column to be retrieved and displayed on the display array. The where clause criteria and the field labels for the two screen fields to be displayed in the display array are also passed to the function. The function returns data from the two columns in the row where the cursor was resting when the user pressed the ACCEPT key.

```
database stores5

main
define xchar1 char(20), xchar2 char(20),xchar3 char(20)

call zoom2( "systables", "tabname", "tabid"," tabid > 99",
                "Table Name", "ID" )
    returning xchar1, xchar2

display "returned ",xchar1 clipped, " | ", xchar2 clipped
sleep 1

call zoom2( "syscolumns,systables ",
            "colname", "coltype",
            " systables.tabid = syscolumns.tabid and ⇐
     systables.tabname = 'syscolumns'",
            "Column Name", "Type" )
    returning xchar1, xchar2

display " returned ", xchar1 clipped, " | ", xchar2 clipped
sleep 1

end main
```

The zoom2 function receives its parameters much as the zoom1 function did. The only difference between this function and the zoom1 function is that two fields are retrieved from the database instead of one. This is reflected in the foreach statement, which loads both fields, the display statement, which will display both field labels to the

screen and the `return` statement, which will return both values the user has selected. The source code for this function is shown below.

```
function zoom2(pTable, pCol1, pCol2, pWhere,
                  pLabel1, pLabel2 )
define pTable   char(60)
define pCol1,
       pCol2    char(60)

define pLabel1,
       pLabel2  char(20)

define pDis_arr array[100] of
   record col1 char(20),
          col2 char(20)

   end record

define pWhere char(100)
define pcount smallint

define Arr_Limit smallint
define qstr char(1000)

let Arr_Limit = 100

let qstr = "select ",pCol1 clipped
if pCol2 IS NOT NULL then
   let qstr = qstr clipped, ",", pCol2 clipped
end if

-- create the statement
let qstr = qstr clipped, " from ", pTable clipped

if pWhere IS NOT NULL then
    let qstr = qstr clipped," where ", pWhere clipped
end if

whenever error continue
prepare _curs1 from qstr
open _curs1

if sqlca.sqlcode < 0   or
```

```
   sqlca.sqlcode = 100  then
   error sqlca.sqlcode using "Error: <<<< on open. "
   sleep 1
   return "", "",  ""
end if

-- load the data into the array
let pcount = 1
foreach _curs1 into pDis_arr[pcount].col1,
                    pDis_arr[pcount].col2

    if pcount + 1 > Arr_Limit then
       error "Too many rows."
       exit foreach
    end if
    let pcount = pcount + 1
end foreach
let pcount = pcount - 1

if pcount = 0  then

   error "No rows found."
   sleep 1
   return  "", ""

end if

open window _wif1 at 10,10
     with form "zoom2"
     attribute ( border,
                 message line last,
                 form line first )

display pLabel1,
        pLabel2
   to   label1,
        label2

 attribute (reverse)
call set_count(pcount)
message "Move cursor to row and press <Esc> to select."
     attribute( reverse )
```

```
display array pDis_arr to zoom2_rec.*

let pcount = arr_curr()

close window _wif1

return pDis_arr[pcount].col1, pDis_arr[pcount].col2

end function
```

input array EXAMPLES: GENERIC input array FUNCTIONS WITH ON-SCREEN SORTING

The input array statement involves coding preparations similar to that of the display array statement, but provides input facilities in addition to the scrolling and display capabilities of the display array statement. A series of statements are provided to allow actions to be trapped before and after a field. Display can be directed to a specific field, but no facility is currently provided to explicitly move the input operation from one row to another; this only happens implicitly when the cursor leaves the current row after the return key has been pressed, or as a result of the action of the up or down arrow key.

The generic input functions presented here accept parameters for the number of rows for the user to input and the field labels to display on the input screen. The user is then allowed to input data to the array. If, while inputting to the screen, the user presses the Control-T key, the data being displayed is sorted and then the rows displayed on the screen are redisplayed. If the user attempts to move beyond the number of rows allowed in the array, an error message is displayed and the user is forced to remain in the current row.

The screen form for the single-row generic input array function is identical to that used for the display array function, with the exception that the screen record is named differently. The sample code that demonstrates the use of this function must declare a global array of records to be used for the input. This is the only method of *passing* an array back to a calling function in INFORMIX-4GL.

These functions can be used to either modify an existing array of data or input a new array of data. It is the programmer's responsibility to see that the program array is in the correct state for calling the function.

In the code example provided below, the array has not been loaded with data, so it must be initialized. The main module first calls the init_arr1 function to initialize the array elements to NULL. (Obviously, if the array elements had been loaded with data,

this function would not have been called.) The `iarr1` function is then called with parameters indicating that 10 elements and the name of the field label should be input.

```
database formonly
screen
{

[l1                    ]
[f1                    ]
[f1                    ]
[f1                    ]
[f1                    ]
[f1                    ]


}

attributes
l1 = formonly.label1;
f1 = formonly.field1;


instructions

screen record iarr1_rec[5] ( field1 )

--------------------------------------------------------------------
iarr1.4gl
--------------------------------------------------------------------

globals

define iarr1_inp
   array[50] of  record
       field1 char(20)
  end record
define _ARRLIMIT smallint
end globals

main
define n,pcount smallint

call init_arr1( 50 )
```

```
let pcount=iarr1( 10, "Names" ) -- perform the input array ⇐
    operation

-- display the results back to the screen
for n =1 to pcount
   display "count ",n, " value ", iarr1_inp[n].field1
end for

prompt "Press return to continue: " for n

end main
```

The `iarr1` Function

The `iarr1` function first opens and displays a window to the terminal screen. It then displays the field label to the screen and calls `set_count` with the number of elements passed into the function as parameters. (The `set_count` function will inform the `input array` facility of the initial size of the program array.) The `input array` statement is then executed using the `without defaults` clause, which will preserve any data currently in the program array variable.

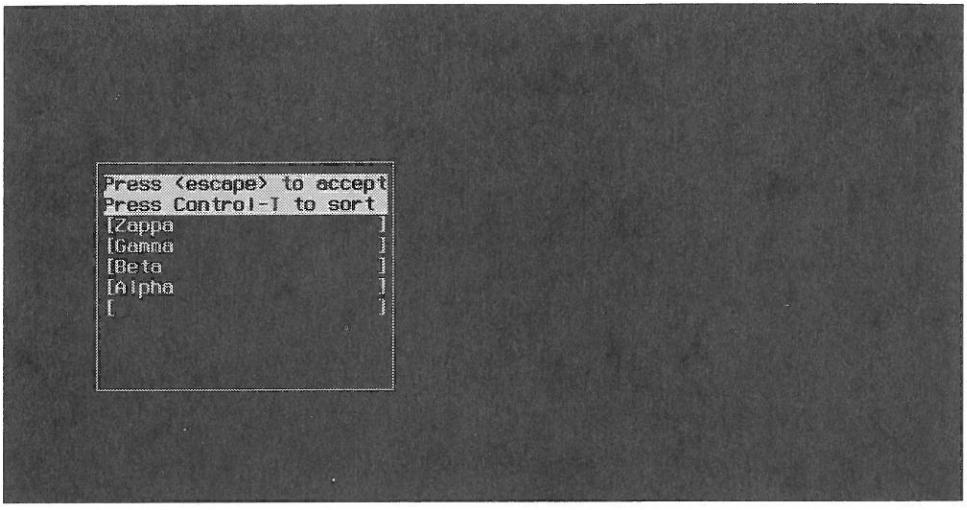
Sorting Data on the Screen

Within the input array statement, the `on key` clause is used to trap the Control-T key. This section of code performs the sort of data currently loaded in the program array. The sort is performed using the `bsort` function, explained in Chapter 14.

Once the `bsort` function has been called, the sorted data for the rows currently on the screen must be displayed. This is accomplished by first determining the current row of the program array that is being input as returned by the `arr_curr` function, and the current screen line where input is being performed as returned by the `scr_line` function. Since we know that the highest screen line that can be displayed on this screen is five, a `for` loop is executed to display all data from the current screen row to the fifth, or *bottom*, row of the screen. Within the `for` loop, the program variable element, as defined by the `x` variable, is used to display the program array element to the screen array element as defined by the n variable, the screen array control element.

The next `for` loop is used to redisplay all of the data from the current screen row to the *top* of the screen. Before the loop, the variable `x` is reset to the current program array element. Within the `for` loop, rows of the program array are displayed to the screen down to screen row one. Note that the `step` used for this loop is a negative one, thus causing the `for` loop to decrement its counters down to one.

Figure 7-3 The iarr1 Function (sorted)



Enforcing the End of an Input Array

The next block of code is used to enforce the end of the `input array` elements. By default, the `input array` will not enforce the end of the program array as defined by the value passed to the `set_count` function. The `input array` statement will allow the user to enter data up to the defined limit of the program array being used to perform input. In order to enforce the end of an `input array` different from the defined size of the program array, special code needs to be executed within the `after field` blocks of the array record as shown below.

After the final field of the program array record, the current program array row (as returned by the built-in `arr_curr` function) is evaluated. If the next row is moving beyond the number of allowable rows, the limit being enforced, an error message will be displayed. However, this will only occur if the user is moving *down* and attempting to enter an illegal row. If the user is moving up or exiting the `input array` loop, they are not attempting to enter an illegal row and the move will be allowed.

The `fgl_lastkey` function returns the last keystroke pressed. This value is evaluated to determine whether or not an arrow key or the ACCEPT key has been pressed. If the value entered is any key other than these, an error message will be displayed and the user will be forced to remain in the same field. Using a `next field` statement that references the current field will force the cursor to remain in the current field.

Once the `input array` statement has been completed, the current window is closed. The `return` statement will then return the row number of the program array row

where the cursor was resting when the ACCEPT key was pressed. The calling function can then use this number to retrieve the correct element from the global program array.

The `init_array` function shown below simply takes the number of elements passed as a parameter and then initializes the global program array up to that limit. The initialize statement will set each element of the array to NULL. The code for the `bsort` function is shown here for completeness, but will not be discussed in this chapter. This function is presented in Chapter 14.

```
function iarr1( array_count, plabel1 )
define array_count  smallint
define plabel1      char(20)
define n,x,num_rows     smallint

open window _wiarr1 at 10,10
    with form "iarr1"
    attribute(border,
              form line first,
              prompt line last )

display plabel1 to label1 attribute( reverse )

--
-- this will make set_count sensible on input array without ⇐
    defaults
-- but it won't enforce the end of the array-the programmer ⇐
    must do that
--
call set_count( array_count )

-- enter input array
input array iarr1_inp without defaults
    from iarr1_rec.*

    on key (Control-T) -- sort the rows
      let num_rows = arr_count( )

      call bsort( num_rows )

      -- redisplay the screen rows
      let x = arr_curr()
      for n = scr_line() to 5
         display iarr1_inp[x].field1 to field1[n]
         let x = x + 1
```

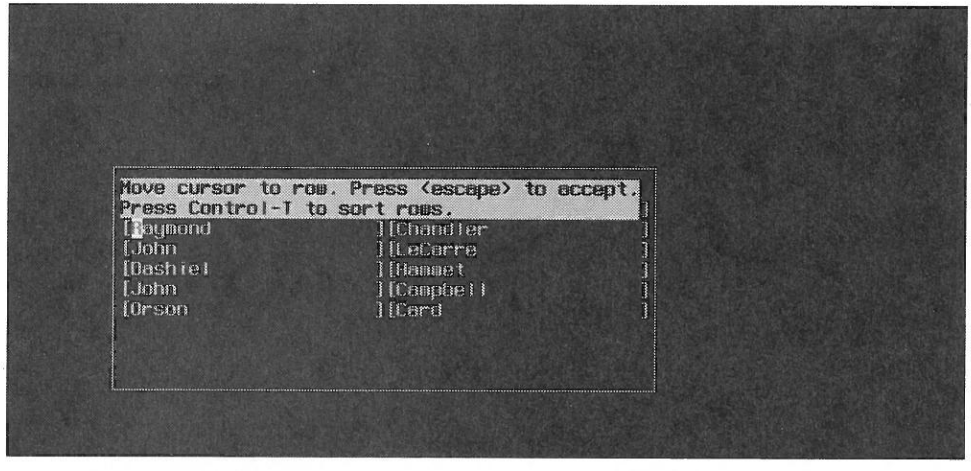
```
      end for
      let x = arr_curr()
      for n = scr_line() to 1 step -1
         display iarr1_inp[x].field1 to field1[n]
         let x = x - 1
      end for

   after field field1   -- enforce the end of the array
      if arr_curr( ) + 1  > array_count  and
         fgl_lastkey( ) != fgl_keyval( "UP" )  and
         fgl_lastkey( ) != fgl_keyval( "ACCEPT" )  then
         error "No more rows."
         next field field1
      end if

end input

close window _wiarr1

return  arr_count( )

end function

---------------------------------

function init_arr1( parrlimit )
define parrlimit, n smallint

let _ARRLIMIT = parrlimit
for n = 1 to _ARRLIMIT

  initialize iarr1_inp[n].field1 to NULL

end for

end function

---------------------------------

function bsort( numelements )
define numelements smallint
define n, swapped  smallint
define tmp char(10)
```

```
let swapped = TRUE
while swapped
  let swapped = FALSE

    for n = 1 to ( numelements - 1 )

       if iarr1_inp[n].field1 > iarr1_inp[n+1].field1  ⇐
    then   # swap them

               let tmp = iarr1_inp[n].field1
               let iarr1_inp[n].field1   = iarr1_inp[n+1].field1
               let iarr1_inp[n+1].field1 = tmp
               let swapped = TRUE
       end if
    end for

end while

end function
```

Generic Input Array Function Using Two Elements

This `function` is virtually identical to the `iarr1` function, except in the handling of the additional parameter. The `iarr2` function accepts parameters for the number of elements to input and field labels to use for the elements. The screen generated by this function is shown in Figure 7-4.

Figure 7-4 The iarr2 Function



The screen form for the `iarr2` function is identical to that used for the `zoom2` function, with the exception of the screen record name. The screen record name for the `iarr2` function is `iarr2_rec`.

```
database formonly
screen
{

[l1                    ][l2                    ]
[f1                    ][f2                    ]
[f1                    ][f2                    ]
[f1                    ][f2                    ]
[f1                    ][f2                    ]
[f1                    ][f2                    ]


}
attributes
l1 = formonly.label1;
l2 = formonly.label2;
f1 = formonly.field1;
f2 = formonly.field2;

instructions

screen record iarr2_rec[5] ( field1, field2 )
```

The `main` Program Block

The sample code for the `iarr2` function is shown below. The `globals` section defines an array of records with two elements, both of which are character strings. Since INFORMIX-4GL converts freely between simple data types, these character strings can be used to input data of any data type.

In the main program block, the `init_arr2` function is called to initialize the global record array before input. If the `iarr2` function were being used for updating data and the global record array were populated with data before the `iarr2` call, this step would not be necessary. If the `iarr2` function is being used for entering new data, the array must be initialized since the function is using the `input array without defaults` syntax, which assumes that the program array holds sensible values and does not initialize the array.

The `iarr2` function is then called with the number of elements to input and the field labels for the two fields being input. The function returns the program array element where the cursor was resting when the user pressed the ACCEPT key.

```
globals

define iarr2_inp
```

```
   array[50] of  record
       field1 char(20),
       field2 char(20)
  end record
define _ARRLIMIT smallint
end globals

main
define n,pcount smallint

call init_arr2( 50 )
let pcount = iarr2( 10, "First Name ", "Last Name" )

message "Entered ",pcount
for n =1 to pcount

   display "count ",n, " value ", iarr2_inp[n].field1

end for

prompt "Press return to continue: " for n
end main
```

The `iarr2` Function

The `iarr2` function will first open a window and display the field labels to the screen. Next, the `set_count` function is called. This will supply an initial entry for the `arr_curr` function to use to maintain an internal count of the number of values in the program array.

The `input array` statement is then executed. Within the `input array` block, the Control-T key is trapped and used to sort the current number of elements in the program array. The current number of elements is determined by first calling the `arr_count` function to get the count.

Once the array has been sorted, the current displayed rows must be redisplayed to reflect the new sort order. This is accomplished using two `for` loops. The first will redisplay the rows from the current screen row to the bottom of the screen. The second will display from the current screen row to the top of the screen.

The next control block is an `after field` statement that is used to force the user to stay within a set number of rows in the `input array`. This is managed by determining whether the user is attempting to move beyond the end of the program array by pressing the DOWN key or the RETURN key; if so, an error message is displayed and the user is forced to remain in the current row using the `next field` statement, as shown below.

If the user is pressing the UP key, or the user has pressed the ACCEPT key, they are allowed to exit that field. This section of code must be repeated for each field in the `input array` row, or the user would be able to enter one of these fields, press the DOWN key and move to an invalid row.

(Note that on some platforms and with some versions of INFORMIX-4GL, executing `next field` on the last field of the screen record would cause the cursor to move to the next row. One workaround for this is to add a no-entry field as the last field in the screen record, so that the `next field` statement is not really executed on the last field of the screen record.)

After the end of the `input array` statement, the window is closed and the function returns to the row in the program array where the cursor was resting when the user pressed the ACCEPT key.

```
function iarr2( array_count, plabel1, plabel2 )
define array_count  smallint
define plabel1, plabel2     char(20)
define n,x,num_rows     smallint

open window _wiarr2 at 10,10
    with form "iarr2"
    attribute(border,
              form line first,
              prompt line last )

display plabel1, plabel2
   to label1, label2
   attribute( reverse )

--
-- this will make set_count sensible on input array ⇐
    without defaults
--but it won't enforce the end of the array-the programmer ⇐
    must do that
--
call set_count( array_count )

-- enter input array
input array iarr2_inp without defaults
    from iarr2_rec.*

    on key (Control-T) -- sort the rows
```

```
      let num_rows = arr_count( )

      call bsort2( num_rows )

      -- redisplay the screen rows
      let x = arr_curr()
      for n = scr_line() to 5
         display iarr2_inp[x].field1 to field1[n]
         display iarr2_inp[x].field2 to field2[n]
         let x = x + 1
      end for
      let x = arr_curr()
      for n = scr_line() to 1 step -1
         display iarr2_inp[x].field1 to field1[n]
         display iarr2_inp[x].field2 to field2[n]
         let x = x - 1
      end for

   after field field2  -- enforce the end of the array
      if arr_curr( ) + 1  > array_count  and
         fgl_lastkey( ) != fgl_keyval( "UP" )  and
         fgl_lastkey( ) != fgl_keyval( "ACCEPT" )  then
         error "No more rows."
         next field field1
      end if

   after field field1  -- enforce the end of the array
      if arr_curr( ) + 1  > array_count  and
         fgl_lastkey( ) != fgl_keyval( "UP" )  and
         fgl_lastkey( ) != fgl_keyval( "ACCEPT" )  then
         error "No more rows."
         next field field1
      end if

end input
close window _wiarr2

return  arr_count( )

end function
```

The `init_arr2` and `bsort2` Functions

The `init_arr2` function initializes the program array for the total number of elements the array will hold. The function also sets the global variable _ARRLIMIT with the number of elements being used; this parameter will be used to limit loops performed on the program array.

The `bsort2` function is passed the number of elements currently in the program array. It then sorts these elements using a bubble sort routine. In this case, the function sorts the array based on the contents of the second field; the first field of the record is ignored. The operation of this function will be covered in detail in Chapter 14.

```
function init_arr2( parrlimit )
define parrlimit, n smallint

let _ARRLIMIT = parrlimit
for n = 1 to _ARRLIMIT

  initialize iarr2_inp[n].* to NULL

end for

end function

------------------------------------------------

function bsort2( numelements )
define numelements smallint
define n, swapped  smallint
define tmp record
    field1 char(20),
    field2 char(20)
   end record

let swapped = TRUE
while swapped
  let swapped = FALSE
    for n = 1 to ( numelements - 1 )

       -- sort just on the 2nd field of the record
       if iarr2_inp[n].field2 > iarr2_inp[n+1].field2  ⇐
    then   # swap them

               let tmp.* = iarr2_inp[n].*
```

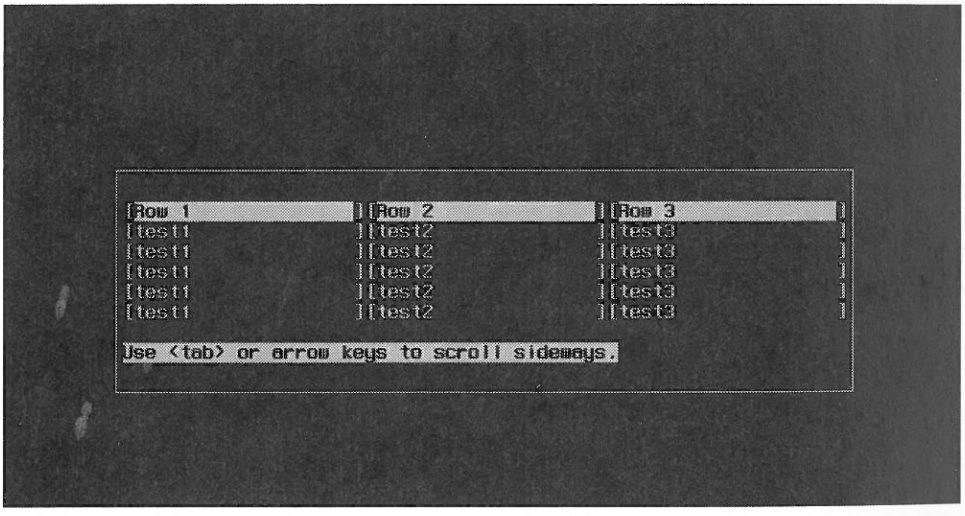
```
                let iarr2_inp[n].*   = iarr2_inp[n+1].*
                let iarr2_inp[n+1].* = tmp.*
                let swapped = TRUE

        end if
    end for

end while

end function
```

Sideways Scrolling `input array` Function

The `input array` statement does not automatically provide sideways scrolling. To get the functionality of sideways scrolling, some additional programming is required. The following example shows what is necessary to program an `input array` application that allows sideways scrolling. Figures 7-5, 7-6 and 7-7 show a sample of a sideways scrolling `input array` application.

This example implements sideways scrolling using two program arrays. One program array is used with the `input array` statement; the other represents the full view of the data being updated with the `input array` statement and may have more columns than the program array that interacts directly. It is the constant effort to keep these two arrays synchronized that concerns much of the program logic.

Figure 7-5 Sideways Scrolling Function Before Right Scroll



When the cursor reaches the right-most column in the `input array`, the user can either press the right arrow key until the end of the current field is reached and the cursor attempts to leave the field, or the user can press the tab key which will cause the application to immediately attempt to move into the next field. This will cause the application to scroll the screen to the right.

When the cursor reaches the far left, if the left arrow key is detected the application will shift the rows to the left. Upward and downward scrolling occur as they would in any other `input array`.

Implementation

The two arrays, the `inp_arr` array and the `disp_rec` array, are synchronized using the left-most element of the screen array. The program array will track which element is being displayed in this position and then shift the screen array to the right or left as needed. Two functions are called to handle the data shift: `load_rec` and `save_rec`.

Figure 7-6 Sideways Scroll After Scrolling Two Positions

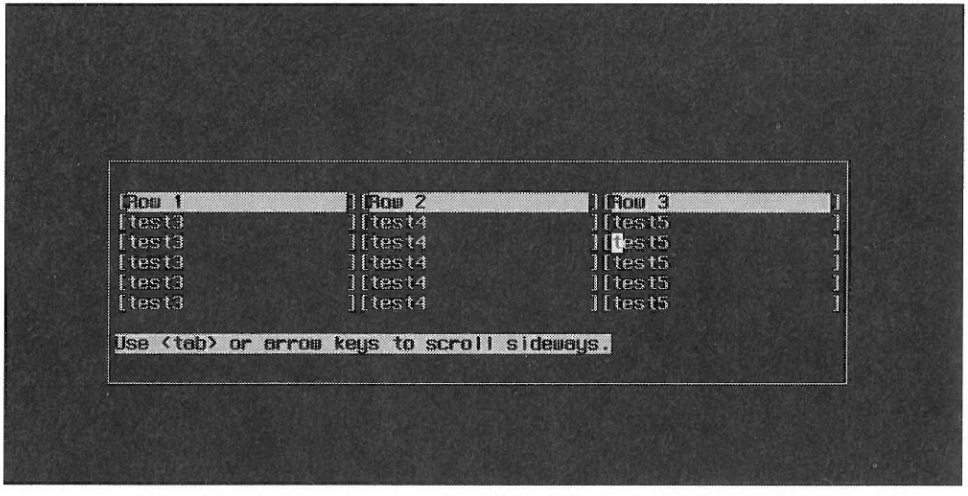
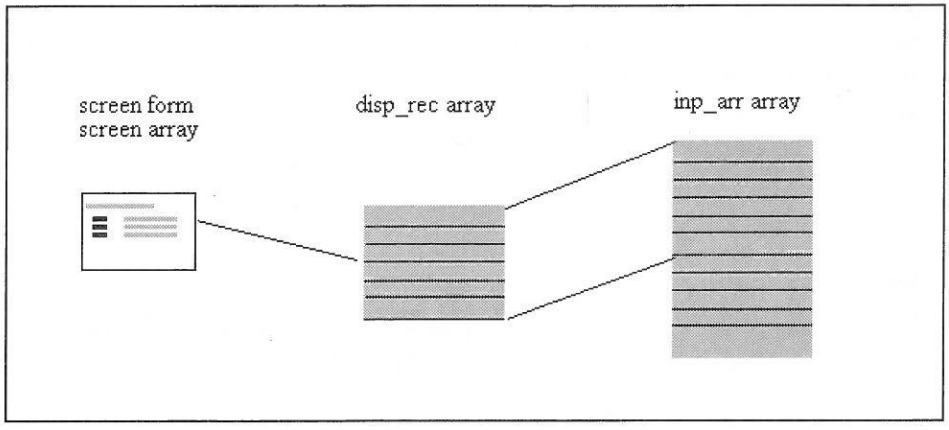


Figure 7-7 Array Synchronization in the Sideways Scroll Function



The `save_rec` function is called before the `load_rec` function. The `save_rec` function will save the current contents of the `disp_rec` array in the `inp_arr` array. This is necessary in case there have been any modifications to the contents of the `disp_rec` array by the user. Also, this function must be called before the `load_rec` function since the `load_rec` function will alter the contents of the `disp_rec` array.

Next, the `load_rec` function is called and passed the current column number (displayed in the left-most array element on the screen) and the next column to be displayed at that screen location. This set of parameters allows this function to be called to move more than one cell if desired, though that is not done in this application.

The `load_rec` function will then perform the shift of the arrays. It will load the data currently in the `inp_arr` array, the master array containing the actual program data, into the `disp_rec` array, the array being manipulated by the `input array` statement.

PROGRAM `main` AND `globals` SECTION

The code implemented for this application is virtually identical to the code implemented for the `input array` examples in this chapter. The difference lies in the code executed when the application detects an attempt to move in a sideways direction.

A number of globals are defined to implement the application. A two-dimensional array, the `inp_arr` array, is created to store up to five columns and 50 rows of data. This will store the *master* data for the application.

A program array of three elements is then defined to correspond with the three-element screen array. It will be necessary to synchronize this array with the five-column

array whenever the user attempts to scroll. Several constants are also defined and then set early in the `main` program block.

Next, in the `main` program block, the array is loaded with test data which will make it clear that the array has successfully scrolled sideways. Now that everything has been prepared, the `sideways` function is called with four parameters, as follows.

```
let pcount = sideways( 30, "Row 1 ", "Row 2 ", "Row 3" )
```

The first parameter is the number of rows the `sideways` function will allow. The `sideways` function will use this number to enforce the end of the `input array` program array. The next three parameters are the screen labels to display for each column of the screen array. Since three columns will be displayed, three screen labels are passed into the function. The `sideways` function returns to the program array element where the cursor was when the user pressed the ACCEPT key.

```
globals
define inp_arr array[50,5] of char(20)
define ARRAY_ELEMENTS,
       SCREEN_ELEMENTS,
       SCREEN_COLS,
       PROG_ARR_COLS   smallint
define disp_rec array[50] of record
     field1 char(20),
     field2 char(20),
     field3 char(20)
  end record

end globals

main
define n,x,pcount smallint

let ARRAY_ELEMENTS  = 50
let SCREEN_COLS     = 3
let SCREEN_ELEMENTS = 5 -- screen array elements
let PROG_ARR_COLS   = 5
-- call init_arr3( ARRAY_ELEMENTS )

for n =1 to ARRAY_ELEMENTS

  for x = 1 to PROG_ARR_COLS
    let inp_arr[n,x] = x using "test<<"
  end for
```

```
end for

let pcount = sideways( 30, "Row 1 ", "Row 2 ", "Row 3 " )

end main
```

The `sideways` Function

The `sideways` function is passed parameters for the total number of rows and the three field labels that will be displayed on the input array screen. The function will first open the window and display the three field labels that have been passed into the function to the screen. The `set_count` function is called and passed the number of elements parameter that has been passed into the function. The current column flag that is used to keep track of the array's current position (the `curr_col` variable) is set to one, since the program will start with the first program array element in the left-most position.

The `input array` Statement

The `input array` loop is then started. It is executed using the `without defaults` clause since the program array has already been loaded with values.

In the `after field` block for the fields `field1`, `field2` and `field3`, the end of the program array is enforced. If the current program array row plus one, which is the row where the cursor may be moving, is greater than the number of program array rows allowed, the end of the array must be enforced.

The remaining Boolean clauses determine where the user is attempting to move. If the user is moving up, the move should be allowed since the cursor will not be moving beyond the end of the program array but staying within its bounds. If the user has pressed the ACCEPT key, the user is not attempting to move beyond the end of the `input array` statement block but is in the process of leaving the input loop. These two keys, the ACCEPT key and the UP key, will therefore be allowed. If the user has not entered one of these keys, the move will be prohibited by executing the `next field` statement and forcing the cursor to stay in the current row.

Scrolling the Array Sideways

The next set of `if/then` statements determine whether or not the cursor is in `field3` or `field1`. If the cursor is in either of these fields, the application must deal with the contingency that the user is attempting to scroll the array to the right or left.

The application must first determine in which direction the user is attempting to scroll the array. Before executing the `if/then` statements, the moved flag is set to FALSE. When the application must determine whether or not the screen array must be redrawn, it will examine the value of this flag.

The `infield` function returns true if the cursor is in the field name passed to the function. If the statement `infield( field3 )` tests true, the cursor is in the `field3` screen field. If the cursor is in `field3`, it is in the right-most column of the screen array. The application must determine whether or not the user is attempting to scroll the screen to the right. This is determined by calling the `fgl_lastkey` function and comparing the value of the last key pressed against the TAB key or the RIGHT (right arrow) key. If the user has pressed either of these keys, it is asserted that the user is attempting to scroll the array to the right.

In order to scroll the array to the right, the value of the current program array must be saved. All data in the program array, on or off the screen, will be saved. This is accomplished with a call to the `save_rec` function.

Next, the data in the array element being moved onto the screen must be loaded into the program array. This is accomplished using the `load_rec` function. This function is called with the value of the current screen column and the current screen column plus one, which indicates to the `load_rec` function that the user is trying to scroll the screen to the right. Finally, the `moved` flag is set to TRUE, indicating that the screen data has been moved and the visible screen array must be redrawn.

When `load_rec` has completed loading the new screen of data, it returns the current column (stored in the `curr_col` variable). This is the value of the `inp_arr` array element displayed in the left-most column of the screen array.

The next `if/then` statement block calls the `infield` function to determine whether or not the user is in field `field1`. This is the left-most field on the screen. If the user is in this field, the `fgl_lastkey` function is called to determine the last key pressed. If the user pressed the LEFT key, the application code asserts that they are trying to move to the left. The `save_rec` function is called to save the data currently contained in the program array. Next, the `load_rec` function is called with parameters for the current screen column and the current screen column less one. These parameters indicate to the `load_rec` function that the user is attempting to scroll the screen to the left. Finally, the `moved` flag is set to the value TRUE, indicating that the screen has been moved and the visible screen array must be redrawn.

Redrawing the Visible Screen Array

The next section of code redraws the visible portion of the screen array, but it only performs this function if necessary. The `moved` flag indicates that the screen must be redrawn, so this flag is evaluated before the screen redraw code is executed. If the `moved` flag is set to TRUE, the screen will be redrawn.

First, the current program array position is stored. Next, the current screen array position is stored as the starting index variable for the `for` loop. A `for` loop is then executed from the current screen line to the total number of screen lines allowed (stored in the SCREEN_ELEMENT *constant*). For each iteration of the `for` loop, the

appropriate program array element is displayed to the appropriate screen array element. This loop will process all screen array elements from the current array element to the end of the current screen array.

However, this will only take care of the current screen row to the bottom row of the screen. The screen array must still be redrawn from the current screen array position to the top of the screen. This is accomplished in the next `for` loop, which will process from the current screen array row to screen array row one, which represents the top of the screen. Note that the `step` value for this loop is negative one.

After redisplaying the screen, the cursor is forced to remain in the current row using the `next field` statement. This is intended to emulate the behavior of a spreadsheet, which would leave the cursor in the current row after scrolling to the right or left. (Note that some versions of INFORMIX-4GL would move the cursor to the *next* row on executing the `next field` statement on the right-most element of the array row. A workaround to this functionality is to add a `noentry` column as the rightmost element in the array row.)

After the `end input` statement, which ends the `input array` block, the `save_rec` function is called to save the data currently on the screen. At this point, the `inp_arr` array is completely synchronized with the `disp_rec` array. The final statement of the `sideways` function closes the display window. The function then returns to the calling function.

At this point, the master program array, the inp_arr program array, holds the modified data.

```
function sideways( pcount, label1, label2, label3 )
define pcount smallint
define label1, label2, label3     char(20)
define moved,n,x,num_rows     smallint
define curr_col smallint

open window _siarr3 at 10,10
    with form "iarr3"
    attribute(border,
              form line first,
              message line last - 1  )

display label1, label2, label3
   to label1, label2, label3
   attribute( reverse )

message "Use <tab> or arrow keys to scroll sideways."
    attribute (reverse)
```

```
--
-- this will make set_count sensible on input array ⇐
    without defaults
-- but it won't enforce the end of the array - the ⇐
    programmer must do that
--
call set_count( pcount )

-- load the display record
let curr_col = 1
call load_rec( curr_col, 1, 50 ) returning curr_col

-- enter input array
input array disp_rec
      without defaults
      from iarr3_rec.*

   after field field1, field2, field3  -- enforce the end ⇐
    of the array
      if arr_curr( ) + 1  > pcount  and
         fgl_lastkey( ) != fgl_keyval( "UP" )  and
         fgl_lastkey( ) != fgl_keyval( "ACCEPT" )  then
         error "No more rows."
         next field field1
      end if

      -- do sideways scrolling, if necessary
      let moved = FALSE
      if infield(field3) then
         if ( fgl_lastkey() = fgl_keyval("RIGHT") ) or
            ( fgl_lastkey() = fgl_keyval("TAB") )    then
            call save_rec( curr_col, ARRAY_ELEMENTS )
            call load_rec(curr_col, curr_col + 1, ARRAY_ELEMENTS)
                returning curr_col
            let moved = TRUE
         end if
      end if
      if infield( field1 ) then
            if fgl_lastkey() = fgl_keyval("LEFT")  then
               call save_rec( curr_col, ARRAY_ELEMENTS )
               call load_rec( curr_col, curr_col - 1,
                              ARRAY_ELEMENTS )
                  returning curr_col
```

```
            let moved = TRUE
          end if
        end if

      if moved  then  -- redisplay the screen
         -- redisplay the screen rows
         let x = arr_curr()
         for n = scr_line() to SCREEN_ELEMENTS
             display disp_rec[x].field1 to field1[n]
             display disp_rec[x].field2 to field2[n]
             display disp_rec[x].field3 to field3[n]
             let x = x + 1
         end for
         let x = arr_curr()
         for n = scr_line() to 1 step -1
             display disp_rec[x].field1 to field1[n]
             display disp_rec[x].field2 to field2[n]
             display disp_rec[x].field3 to field3[n]
             let x = x - 1
         end for
         if infield(field1) then
            next field field1
         end if
         if infield(field3) then
             next field field3
         end if
      end if

end input

call save_rec( curr_col, ARRAY_ELEMENTS ) -- save what ⇐
    is now on the screen

close window _siarr3

return  arr_count( )

end function
```

The `init_arr3` Function

The `init_arr3` function is used to initialize the array used in the sideways scrolling function. It merely initializes all of the array elements using two nested loops. It then returns to the calling function.

```
function init_arr3( parrlimit )
define parrlimit, n,x smallint

for n = 1 to parrlimit
  for x = 1 to SCREEN_COLS
    let  inp_arr[n,x] = ""
  end for
end for

end function
```

The `load_rec` and `save_rec` Function

The `load_rec` and `save_rec` functions are used to keep the program array and master data array synchronized. The program array is the array used to interface with the `input array` statement. The data array contains the complete array of data to be manipulated by the user. It is the job of these two functions to see that all data input to the screen array makes its way to the master data array.

The `load_rec` function takes as parameters the current row element in the left-most position in the screen array, the next array position to be displayed in the left-most element and the number of elements to process.

The function will first validate the move to be sure it is sensible. This is determined by subtracting the next element plus one from the number of program array columns and validating that it is less than the number of screen columns.

```
if ( PROG_ARR_COLS - next + 1 ) < SCREEN_COLS  then
   let next = 1
end if
```

If the program array has five columns and the user is requesting to move to column 6, the result of this subtraction would be -1. Since the number of screen columns is five, this validation would test false. If the validation fails, the next row is set to one. This yields program behavior that allows the user to move the cursor to the right-most column, attempt to move to the right and see the array wrap around to the first position — functionality similar to that of a spreadsheet.

If, for some reason, the next element parameter (`next`) is less than zero, it is set to one. The `for` loop is then started to move the `inp_arr` array data to the `disp_rec` array. The `for` loop will execute from the first element to the total number of elements (the parameter passed into the function). For each iteration of the `for` loop, the appropriate `inp_arr` array element is assigned to the `disp_rec` field using the following assignments.

```
let disp_rec[n].field1 = inp_arr[n,next]
let disp_rec[n].field2 = inp_arr[n,next + 1]
let disp_rec[n].field3 = inp_arr[n,next + 2]
```

When the `for` loop has completed, the function will return the `next` element value. This is the next element that was passed into the function, but since the function logic may have changed the value, it is significant to return this number.

The `save_rec` function will save the current values in the `disp_rec` array in the `inp_arr` array. It simply uses a `for` loop for one to the number of elements value passed into the function. The assignments are the reverse of those shown above for the `load_rec` function. When complete, this function merely returns to the calling function.

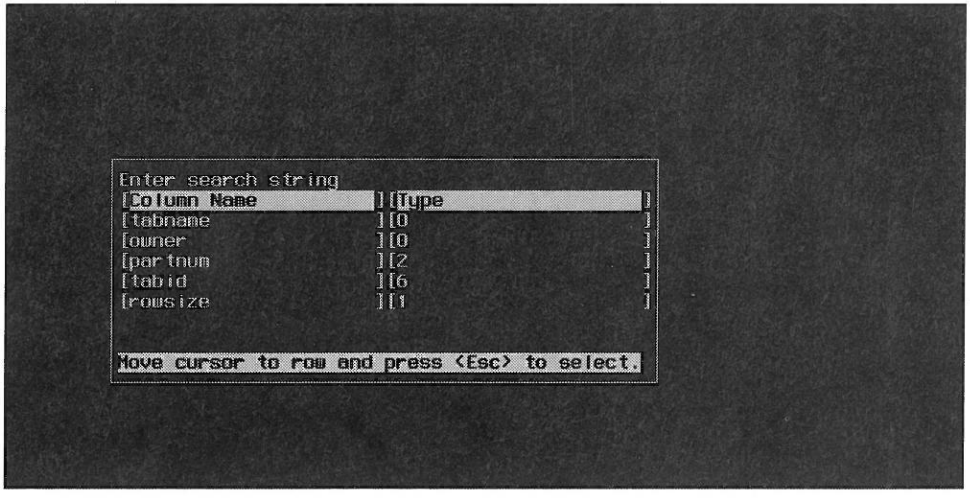
```
function load_rec( curr, next, elements )
define curr, next, elements smallint
define n    smallint

-- make sure 'next' parameter is sensible
if ( PROG_ARR_COLS - next + 1 ) < SCREEN_COLS  then
   let next = 1
end if
if next <= 0 then
   let next = 1
end if

-- load the new elements
for n = 1 to elements

   let disp_rec[n].field1 = inp_arr[n,next]
   let disp_rec[n].field2 = inp_arr[n,next + 1]
   let disp_rec[n].field3 = inp_arr[n,next + 2]

end for

return (next)
```

```
-----------------------------------

end function

function save_rec( curr_scr_col, elements )
define curr_scr_col, elements smallint
define n    smallint

for n = 1 to elements

   let inp_arr[ n, curr_scr_col ]     = disp_rec[n].field1
   let inp_arr[ n, curr_scr_col + 1 ] = disp_rec[n].field2
   let inp_arr[ n, curr_scr_col + 2 ] = disp_rec[n].field3

end for

end function
```

SEARCHABLE DISPLAY ARRAY

Often, programmers find it useful to be able to allow the user to select a set for a display array and then further search that set for a specific value by entering a search value for the current program array set, as shown in Figure 7-8. Upon finding the value, the user would be allowed to see that value at the top of the display array set. The code below provides an example of just such a display array.

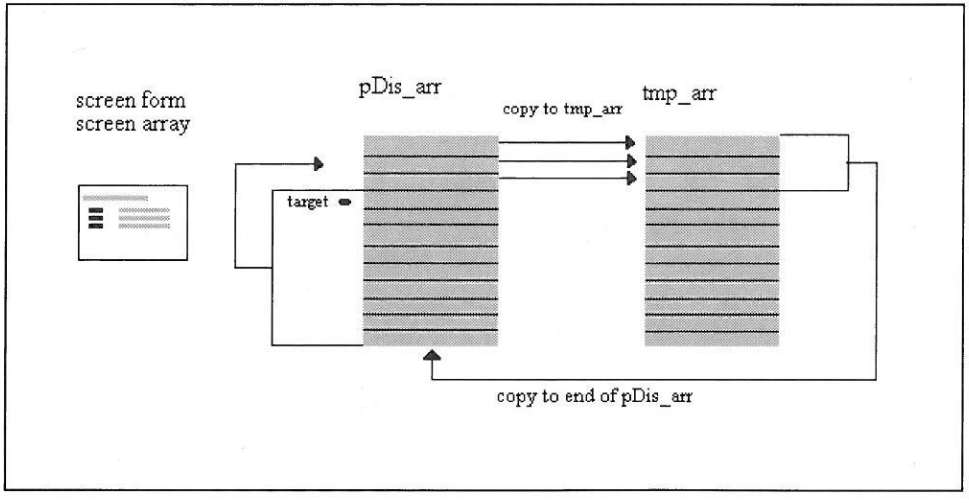
Figure 7-8 Searchable Display Array Function



Implementation

In this example, if the user presses the Control-F key, a prompt message is displayed allowing the user to enter a search string. If the search string is found, that program array element where the search string was found becomes the top element in the array. The program array rows that have previously been *above* that element are shifted to the bottom of the array. The user is then placed back under the control of the display array statement. Figure 7-9 shows this operation.

Figure 7-9 Array Synchronization in the searchit Function



THE searchit FUNCTION

The flow of control for the searchit function is similar to the zoom2 function shown earlier in this chapter. The only difference is in the handling of the display array loop. INFORMIX-4GL does not allow the display array to shift the cursor to a specific array row. The only means of moving the focus of the display array up or down in an array is to rebuild the program array to display with the row to enter at the top of the list, exit the display array, and then execute the display array statement again with the shifted contents of the program array. The same procedure is required with the input array statement.

As with the zoom2 function, the searchit function is called with the table name, the columns to be displayed, the where clause to use to retrieve the data and the two screen labels to display to the screen for the columns.

The retrieval of data, opening of the display window and display of the screen form are identical to those of the zoom2 function. The difference lies in the execution of the display array statement. With the searchit function, the display array statement is executed within a while loop, executed as shown below.

```
    while ( 1 )

      display array pDis_arr to zoom2_rec.*
    ...


      end display

      if fgl_lastkey() = fgl_keyval( "ACCEPT" ) ⇐
      then -- user is done
                  exit while
      end if
  end while
  ...
```

This while loop will continue until the user presses the ACCEPT key. The reason for this loop is the need to constantly reenter the display array statement loop with a newly shifted program array.

The display array statement is initially executed using the original state of the program array. When the user presses the Control-F key, a set of statements are executed that will first search the array for the value the user has entered and, if found, will shift the data in the array until the element found is at the top of the array.

When the Control-F key is pressed, the user will first be prompted for a string pattern to find. The string entered is then prepended and appended with the '*' character. A `for` loop is then executed as shown below.

```
for n = 1 to pcount

        if pDis_arr[n].col1 matches sstr  then
           exit for
        end if

    end for
```

The `for` loop scans the program array, starting at the first element and proceeding to the last element. The `if/then` statement takes advantage of the string search capabilities of the INFORMIX-4GL matches clause. If a string is found, the `for` loop is exited; at this point, the value of `n` would be the program array element at which the string pattern was found.

If, after the execution of the `for` loop, the value of `n` is equal to the value of `pcount` and the number of elements in the program array, and the value of `col1` at that array element does not match the search string, then it is asserted that the value was not found and the display array is exited. Ultimately, the flow of control will take the function back into the `display array` statement loop.

After the execution of the `for` loop, if the value of `n` is not equal to `pcount`, it is asserted that the string pattern was found and the element of the string pattern is the value of `n`. At this point, it is necessary to store the elements in the program array that are *above* the target element, the element where the search pattern was found. This is a precursor to shifting the contents of the program array up to place the target string pattern at the top of the array.

Next, a `for` loop is executed from the start of the program array to the `n` element. For each iteration of the loop, the value in the program array is stored in a temporary array, as shown below.

```
-- store what will be shifted down
for x = 1 to n
  let tmp_arr[x].* = pDis_arr[x].*
end for
```

Once these elements have been stored, the program array is shifted up. Another `for` loop is executed as shown below, this time from the start of the array up to the value of `pcount - n`. This will copy only those elements needed and will exit the `for` loop with the value of `x` pointing to the bottom of the original list. It is also necessary to limit

the loop to `pcount - n`, since the right-hand side of the assignment expression is copying the program array element at `x + n`. If the loop were allowed to copy to the value of `pcount`, illegal array subscripts would be referenced.

```
-- scroll program array up
for x = 1 to ( pcount - n )
        let pDis_arr[x].* = pDis_arr[ x + n ].*
end for
```

Now that the array has been scrolled up, it is necessary to copy in the values that have been overwritten when the array was scrolled up. These values were stored in the `tmp_arr` and are now copied into the bottom of the array starting at the value pointed to by the value of `x`. The loop, as shown below, copies from the value of `x` to `x + n`, the number of elements that were originally stored in the `tmp_arr` array variable.

```
for z = x to x + n
      let pDis_arr[z].* = tmp_arr[ z - x + 1 ].*
end for
```

The formula used to reference the array element to be copied is the current `for` loop iteration number stored in the value of `z` less the value of `x`, the starting element. This will yield the current offset less one, which would be one less than the value needed. Therefore, one is added to this value to yield a useful index value. The sole purpose of this formula is not to confuse the reader, but to avoid having to create a second variable to use as an index for the right-hand side of the assignment expression.

At this point, the program array has been successfully shifted up. It is not necessary to execute code to redisplay the program array, since that will happen automatically when the display array is reentered. It is, however, necessary to exit the display array, which is what the next line of the program will do. This will return the flow of control to the `while` loop, which will force flow of control to reenter the `display array` statement.

After the end of the `display array` statement, the last key pressed is evaluated against the ACCEPT key. The purpose of this statement is to allow the user to gracefully exit the `display array`. In the current implementation of this function, the Control-C key will also serve this purpose.

The complete code for the `searchit` function is shown below. Since most of this code has been explained above or in the section on the `zoom2` function, there is no more detailed explanation provided.

```
function searchit(pTable, pCol1, pCol2, pWhere,
                  pLabel1, pLabel2 )
```

```
define pTable   char(60)
define pCol1,
       pCol2    char(60)

define pLabel1,
       pLabel2  char(20)

define pDis_arr array[100] of
   record col1 char(20),
          col2 char(20)

   end record

define tmp_arr array[100] of
   record col1 char(20),
          col2 char(20)

   end record

define pWhere char(100)
define pcount smallint

define Arr_Limit smallint
define qstr char(1000)
define sstr char(40)
define offset,n,x,z smallint

options defer interrupt
let Arr_Limit = 100

let qstr = "select ",pCol1 clipped
if pCol2 IS NOT NULL then
   let qstr = qstr clipped, ",", pCol2 clipped
end if

let qstr = qstr clipped, " from ", pTable clipped

if pWhere IS NOT NULL then
    let qstr = qstr clipped," where ", pWhere clipped
end if

whenever error continue
```

```
prepare _curs1 from qstr
open _curs1

if sqlca.sqlcode < 0   or
   sqlca.sqlcode = 100  then
   error sqlca.sqlcode using "Error: <<<< on open. "
   sleep 1
   return "", "",  ""
end if

let pcount = 1
foreach _curs1 into pDis_arr[pcount].col1,
                    pDis_arr[pcount].col2

    if pcount + 1 > Arr_Limit then
       error "Too many rows."
       exit foreach
    end if
    let pcount = pcount + 1
end foreach
let pcount = pcount - 1

if pcount = 0  then

   error "No rows found."
   sleep 1
   return  "", ""

end if

open window _wif1 at 10,10
     with form "zoom2"
     attribute ( border,
                 message line last,
                 form line first )

display pLabel1,
        pLabel2
   to   label1,
        label2

 attribute (reverse)
```

```
call set_count(pcount)
message "Move cursor to row and press <Esc> to select."
     attribute( reverse )

while ( 1)
let int_flag = FALSE

display array pDis_arr to zoom2_rec.*

    on key (Control-F)  -- search for a string

      call promptit( "Enter search string " ) returning sstr
      let sstr = "*", sstr clipped, "*"
      for n = 1 to pcount
        -- this loop will end when string is found, or n = pcount
         if pDis_arr[n].col1 matches sstr  then
            exit for
         end if

      end for

      if n = pcount  then -- assert we didn't find the string
         error "String not found."
         exit display
      end if

-- else n is the element we want (n marks the spot ;) )

-- shift program array. make selected row the top row

-- store the elements above n
let n = n - 1

-- store what will be shifted down
for x = 1 to n
  let tmp_arr[x].* = pDis_arr[x].*
end for

-- scroll program array up
for x = 1 to ( pcount - n )
   let pDis_arr[x].* = pDis_arr[ x + n ].*
end for
```

```
-- x now points to end of original list
-- copy in what used to be the top and has now wrapped ⇐
     around to the bottom for z = x to x + n
   let pDis_arr[z].* = tmp_arr[ z - x + 1 ].*
end for

exit display

end display

  if int_flag then
     exit while
  end if

end while

let pcount = arr_curr()

close window _wif1

return pDis_arr[pcount].col1, pDis_arr[pcount].col2

end function

-----------------------------------

function promptit( str)
define str char(20)
define retstr char(40)

prompt str for retstr

return retstr


end function
```

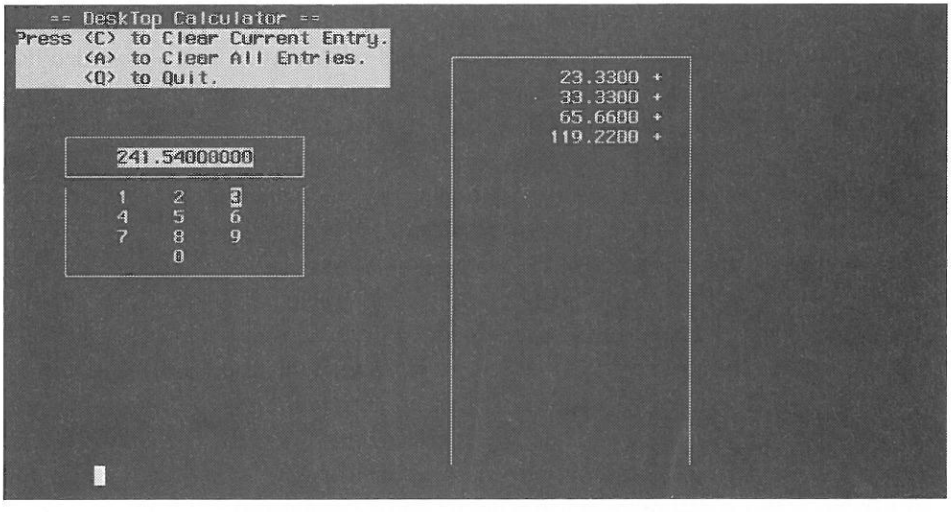
Chapter 8

Additional Screen Examples: Calculator and Calendar

THE CALCULATOR

Another example of a complex screen interaction application is that of a desktop calculator(see Figure 8-1).. Creating a calculator on the screen requires an application that monitors each keystroke, maintains an image of the current number, throws out invalid characters without complaint, and parses and acts on any special non-numeric symbols that are input. While simple screen form input could be considered adequate for this application, the INFORMIX-4GL input statement does not act on **single** character input in a screen form; it waits for a carriage-return or new-line character before returning the input. For this program, interpreting each character as it is input better emulates a desktop character. The only INFORMIX-4GL statement that retrieves a character and returns it to the program immediately is the `prompt for char` statement. This statement, used in conjunction with several processing loops, will provide the basis for this application.

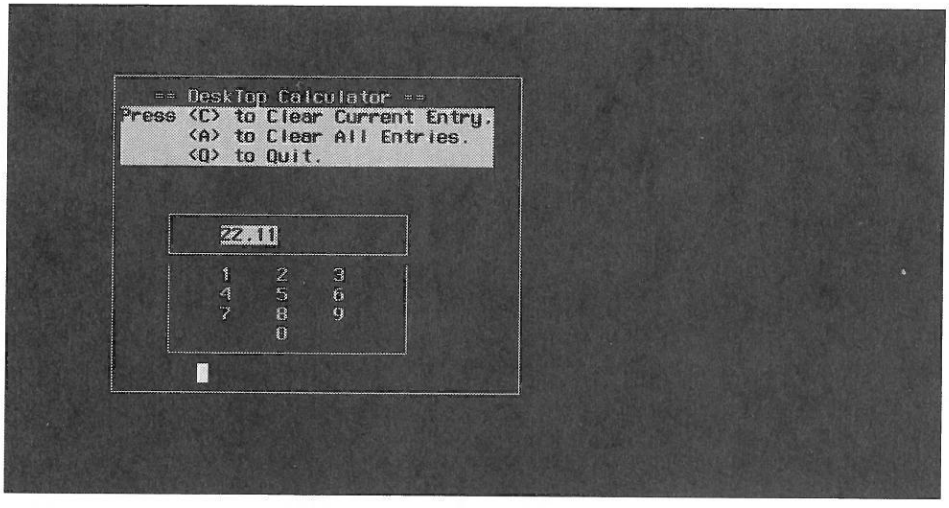
Figure 8-1 Desktop Calculator



This program will display a screen that will resemble a calculator keypad on the left and a window on the right-hand side which will display a *ticker* tape of the currently

entered numbers. Optionally, a single-window version can be used to display just the keypad window as shown in Figure 8-2. A *clear* key is available to clear the currently entered number, and a *clear all* key is provided to clear all entries in the number window and effectively restart the calculator.

Figure 8-2 Desktop Calculator Without Tape



This program will process each character typed on the keyboard. As each character is typed, it will be passed to a processing routine. The processing routine will examine the character; if the character is a number, the number will be displayed in reverse video on the *keypad* and then added to a string being used to hold the current number. If a character representing an operation is entered (+ - / * % E), both the operator character and the current numeric value are loaded into parallel arrays. When an equals (=) character is entered, a function will be called to perform the mathematical operation and return the result. The result will be displayed to the number window.

This program highlights some of the more important features of INFORMIX-4GL. The application converts data types freely by manipulating numbers as character strings and then performing mathematical operations on the contents of those strings. Complicated conversion routines need not be called because this capability is built into the INFORMIX-4GL language. Characters typed must constantly be evaluated for validity within a range of characters; this is performed using the INFORMIX-4GL `if` statement. Data must be displayed at various locations on the screen using different display attributes; this is accomplished with the `display` statement.

The code for this application is shown below. One of the objects in designing this application was to allow the application to be easily integrated into another application. This required a modular design with very few globals and small, compact functions. The

choice of data structures was important. The ability to store arrays in records allowed a significant portion of the data stored by the application to be placed in two separate records. These records are named with a preceding backslash character, thus concealing the name from other global variables in an application into which the calculator may be merged.

An additional aid in the integration of the application is the exclusion of the screen form. Because very little screen input is necessary and the input statement does not provide character processing, the calculator application does not interact with a screen form.

```
define _calc_ctrl record
    val array[20] of decimal,
    op  array[20] of char(1),
    cur_val char(32),
    prev_int char(1),
    eq_flag smallint,
    last_result decimal
end record
```

```
define _scr_ctrl record
    PROMPT_LINE smallint,
    START_ROW  smallint,
    START_COL   smallint,
    NUM_VALS    smallint,
    use_ticker  smallint,
    TICK_ROW    smallint,
    ticker_row, ticker_col,
    total_row,  total_col,
    wind_row,   wind_col   smallint,
    calc_row,   calc_col   smallint,
    clear_number char(15)
end record
```

```
define _calc_counter smallint
```

Variable Definitions

The code shown above is the first section of the calculator application. In this section, two control records are defined to monitor the calculator's operation. Within the `_calc_ctrl` record, the `val` array is defined. This array of twenty decimals is used to hold the values entered by the user. The `op` variable is an array of 20 single characters and is declared to hold the operations performed on each entry; it is kept *parallel* to the

val array. These arrays must be declared with module or global scope because they must be manipulated by multiple functions which cannot take arrays as parameters.

The cur_val character string is declared to hold the currently entered value. This string will be used to build the value that the user has entered. It will continue to build the number until an operator character, a character which signals that a mathematical operation should be performed, is entered. Once an operator character is entered, the cur_val string will be stored and the curr_val number and the operation character will be displayed.

The prev_int variable will be used to hold the last digit entered. As each character is typed, it is displayed in reverse video on the keypad. In order to clear this number when the next character is typed, the current character typed must be saved; and when the new character is typed, the previously entered digit must be redisplayed in normal video before the current character is displayed in reverse video.

Next, the scr_ctrl record is declared with various elements that are used to track the location of calculator components. Because the calculator was written without a screen form, it is necessary to store several screen coordinates. The PROMPT_LINE represents the location of the input prompt. The START_ROW and START_COL represent the screen coordinates for the start of the keypad window. The NUM_VALS element represents the maximum number of values that can be displayed to the keypad window. The ticker_row and ticker_col elements represent the screen coordinates for the ticker window. The total_row and total_col represent the screen coordinates for the location of the current total. The wind_row and wind_col variables represent the screen coordinates for the keypad window, and the calc_row and calc_col represent the coordinates for the start of the calculator keypad. The clear_number element is a 15-character string that is displayed to the number window to clear the previous entry.

Finally, the _calc_counter variable is declared. This counter variable is used to keep track of the current array index into the val and op arrays. Since it must retain its value between calls to the proc_val function, it is declared with module scope.

The main Program and the run_calc Section

Next, the main section of the program begins and, in this case, merely calls the run_calc function. (This demonstrates the ease of integrating this application into another application; only a single function call is required to start the application.) The run_calc function is called with three parameters: the row and column where the calculator should be displayed and a flag indicating whether or not the ticker window should be displayed.

First, within the run_calc function, a series of initializations are performed through a function call to the init_vals function. The prompt line is then set to the value of the PROMPT_LINE record component via the options statement. The record component cur_val is used to hold the current value of the number being input; it is

initialized to NULL, and the `prev_int` variable, which is used to hold the value of the previous integer entered, is also set to NULL.

The `while` loop represents the main loop of this program; all user interaction takes place within this loop. Two primary functions are performed in this loop: a single character is received using the `prompt` statement and this single character is then processed using the `proc_val` function. These two functions will continue to be performed until the user enters the letter "q" or control-C to exit the loop.

Note the use of the `prompt` statement in this loop. If the `prompt` statement contains the clause `for char`, then it will wait for a single character and immediately return it to the program. Since we are trying to emulate a desktop calculator, we want the program to immediately accept and process the number keys pressed on the keypad on the screen. For that reason, the `prompt` statement does not display a prompt string to the terminal; a blank character string is substituted.

```
main
define ret_val decimal

call run_calc( 5, 10, 0 ) returning ret_val
display "Calc returned ",ret_val
sleep 1

end main

--------------------------------

function run_calc( calc_row, calc_col, ticker_disp_flag )
define calc_row, calc_col, ticker_disp_flag smallint
define c char(1)

-- set screen parameters
let _scr_ctrl.calc_row = calc_row
let _scr_ctrl.calc_col = calc_col
let _scr_ctrl.use_ticker  = ticker_disp_flag

call init_vals( )

-- options prompt line _scr_ctrl.PROMPT_LINE
options prompt line last

initialize _calc_ctrl.cur_val to NULL
let _calc_ctrl.prev_int = NULL
let _calc_counter = 1
```

```
call disp_calc()

while TRUE

   prompt "        " for char c
   if upshift( c ) = "Q" then
      if NOT _scr_ctrl.use_ticker  then
         close window _calc1
      end if
      exit while
   else
     call proc_val( c )
   end if

end while

return _calc_ctrl.last_result

end function
```

The `disp_calc` Function

The `disp_calc` function merely displays the screen to the terminal. It first draws a series of boxes to the screen using the `fgl_drawbox` function. This `fgl_drawbox` function takes arguments as follows:

```
fgl_drawbox( number_of_rows,  number_of_columns,⇐
    start_row, start_column )
```

The first two parameters passed represent the size of the box: the number of rows for the box and the number of columns the box will use, in that order. The final two parameters represent where to locate the box: the starting row for the box and the starting column, in that order.

The `fgl_drawbox` function is currently undocumented. If unavailable, this function could be replaced with an `open window` statement using a `border` attribute. Using separate windows to produce the calculator function would require the use of the current window statement to change focus between the keypad window and the ticker tape window.

In this program, all of the parameters to this function are defined as module scope variables and are set in the `init_vals` function. This allows the size of the keypad and the position and size of the ticker tape window to be changed relatively easily by merely changing these parameters within the `init_vals` function.

The ticker window is optionally displayed, depending on the parameters passed to the `run_calc` function. Finally, the keypad is displayed using a `for` loop to count out ten digits from zero to nine. Each digit is passed to the `disp_number` function, which will display the number to the keypad window.

```
function disp_calc( )
define n smallint

if NOT _scr_ctrl.use_ticker then
   open window _calc1 at _scr_ctrl.calc_row,
                         _scr_ctrl.calc_col
                  with 15 rows,
                       35 columns
  attribute ( border )
end if

-- box for the keypad
call fgl_drawbox( 6, 22,
                  _scr_ctrl.START_ROW - 1, ⇐
    _scr_ctrl.START_COL - 5 )

--  box for the keypad window
call fgl_drawbox( 3, 22,
                  _scr_ctrl.wind_row-1, _scr_ctrl.wind_col-5 )

if _scr_ctrl.use_ticker then
  -- box for the ticker window
  call fgl_drawbox( _scr_ctrl.NUM_VALS + 2, 22,
                    _scr_ctrl.ticker_row - 1, ⇐
    _scr_ctrl.ticker_col - 1 )
end if

for n = 0 to 9
   call disp_number( n, 0  )
end for

display "== DeskTop Calculator == " at 1, 4
display "Press <C> to Clear Current Entry." at 2,1
    attribute ( reverse )
display "      <A> to Clear All Entries.  " at 3,1
    attribute ( reverse )
display "      <Q> to Quit.               " at 4,1
    attribute ( reverse )
end function
```

The proc_val Function

The proc_val function performs the bulk of the work for this application. This function will receive the character typed at the keyboard, determine its validity and then perform the necessary processing. As each character is passed to the proc_val function, it is first evaluated; if it is NULL, the default character the plus ('+') sign is substituted. This allows the return key on the keyboard to be used just as the plus ('+') key would be used on an adding machine.

Next, the character is evaluated for its validity using the valid function. This function will merely evaluate the character for being an integer number or an operator character. If the character is not valid, the calculator application will complain quietly, as would happen with a desktop calculator; it will simply continue waiting for the next character to be typed by the user.

The next few lines of code are used to evaluate whether or not the last operation performed was an equals operation. This is determined by evaluating the value stored in the eq_flag element of the _calc_ctrl record. This handles the case where the user has entered an equals character, has evaluated the results displayed on the number window and has now begun typing new characters on the keyboard. In this case, the previously entered numbers should be cleared from the keypad window. If not, the new characters will overwrite the result of the previous equals operation, creating a meaningless number in the keypad window. Therefore, if the eq_flag is set to TRUE, the number pad is cleared; otherwise, the number pad is not cleared.

Next, if the character entered is a number, then it will be appended to the cur_val string, which is used to accumulate the value being typed by the user. This current value is then displayed to the keypad window in reverse video.

Processing the Operator Character

In the next set of statements, the character typed is evaluated to determine whether or not the user has entered an operator character. If an operator character has been entered, the code within this block will be executed.

When the proc_val function is entered, the last typed integer is still highlighted in the keypad window. Since an operator character is entered, the last typed number should be cleared. This is accomplished through a call to disp_number. The disp_number function is called with the variable prev_int, which contains the value of the last typed integer. The disp_number function will clear that value by displaying the number to the correct location on the keypad normal video. The next argument passed is the value to display in reverse video (in this case, it's a zero); the function call will display zero in reverse video on the keypad.

Next, the current value and current operators are stored in parallel arrays, and the display value is set to the current value with the assignment statement.

```
let disp_val = cur_val
```

If the current value is NULL, then `proc_val` returns at this point. The current value is displayed in the keypad number window. On the following line, the same number is displayed in the ticker tape window. The ticker row is incremented, and the current value, the character typed and the previously typed integer are initialized by being set to NULL. The parallel array index variable `_calc_counter` is incremented, and the keypad window is cleared by displaying the `clear_number` string to the location on the screen. At this point, the code returns from this function to cleanly avoid executing any of the other code in the function.

Equals Operation

The next `if/then` statement block evaluates the character typed to determine whether or not the user has entered an '=' character. If the user has, it is assumed that the user wants to complete the operations that have been entered and look at the total in the keypad number window.

First, the `eq_flag` is set to true. This is a static integer variable which will indicate on the next call into this function that the previous operation was an equals operation, and therefore, the number window needs to be cleared when the user begins typing characters on the keypad. This behavior mirrors that of a desktop calculator.

The following lines examine the index variable `i` to determine whether or not the user has entered any values. If the value of `i` is one, the user has not entered any values, and since no operations can be performed, the function merely returns. In keeping with the functionality of a desktop calculator, no error message is displayed; the function merely returns.

Next, processing moves serially through the list of values and their corresponding operators. Within a `for` loop, for each set of entries in the list, the `do_op` function is called with parameters of the currently indexed value in the value list (`val[n]`) and the next value in the value list (`val[n+1]`) and the corresponding operator for the first value passed (`op[n]`). The result is accumulated in the result variable (`result`).

Upon completion of the `for` loop, the result is displayed in the keypad window. The variables that are used to keep track of what the user is typing are then initialized: the current value variable (`cur_val`) is initialized, the currently typed character variable (`c`) is initialized and the previously typed integer variable (`prev_int`) is initialized. Since this represents the completion of this `if/then` block and no more processing is needed, the function returns at this point.

Clearing the Number Window with the *Clear* Option

The next `if/then` block examines the character typed to determine whether or not the user has requested to clear the current number window by typing the 'c' character. If the user has requested to clear the number window, this block of code is executed. This user option is designed to simply clear the currently entered number; all previously

entered entries will remain. This option has the functionality of a *clear* option on a desktop calculator.

If the user wants to clear the number window, the clear number string variable (`clear_number`), which is a string of blanks the exact width of the number window, is displayed to the number window. The current value string is then initialized (since the user has requested that the current entry be cleared) and, since no more processing is required, the function returns at this point.

The *Clear All* Option

The *clear all* option performs the function that is usually executed by depressing the C key twice on a desktop calculator (or on some calculators, pressing the CA key once) and is executed by pressing the A key. When the A key is entered, the `init_vals` function is called to initialize all of the variables associated with monitoring user input. A `for` loop is then executed to clear all of the entries in the ticker window. The keypad window is then cleared, the ticker row variable is reset and other variables associated with the program's operation are initialized. When all the work is completed, the function returns.

All Other Contingencies

If no other operations need be performed, then the `proc_val` function will clear the previously typed digit on the keypad and display the currently typed integer number in reverse video on the keypad. It will store the currently typed integer number as the previous integer number (for the next call to this function) and return.

```
function proc_val( c )
define c char(1)
define n,z,e smallint
define disp_val char(32)
define result decimal

if c IS NULL then
   let c = "+"      -- the default operator character
end if

if NOT valid( c )  then
   return
end if


-- if last op was equals or not using the ticker widow
-- then clear the number window
```

```
if _calc_ctrl.eq_flag  or
   (NOT _scr_ctrl.use_ticker and _calc_ctrl.cur_val is NULL )⇐
       then

   display _scr_ctrl.clear_number
     at _scr_ctrl.wind_row, _scr_ctrl.wind_col
end if

--
-- display the current number val
--
if ( c >= "0" and c <= "9" ) or ( c = "." ) then
  let _calc_ctrl.cur_val = _calc_ctrl.cur_val clipped, c
end if
let disp_val = _calc_ctrl.cur_val
display _calc_ctrl.cur_val clipped
      at _scr_ctrl.wind_row, _scr_ctrl.wind_col
        attribute( reverse)

--
-- user is done entering the line
-- display the number and increment the ticker
--
if c = "+" or c = '-'  or c = "*" or c = "/"  or
   c = "%"  or upshift(c) = "E"          then

   call disp_number( _calc_ctrl.prev_int, 0 )  # clear ⇐
     the number pad

   let _calc_ctrl.val[_calc_counter] = _calc_ctrl.cur_val
   let _calc_ctrl.op[_calc_counter]  = c

   let disp_val = _calc_ctrl.cur_val

   if _calc_ctrl.cur_val IS NULL then
      return
   end if

   display _calc_ctrl.cur_val clipped
         at _scr_ctrl.wind_row, _scr_ctrl.wind_col
            attribute( reverse)

   if _scr_ctrl.use_ticker   then
```

```
            if _scr_ctrl.ticker_row = _scr_ctrl.NUM_VALS then
         error "Too many values."
      else
         display _calc_ctrl.val[_calc_counter]
                 using "###,###,###.####", " ",c
             at _scr_ctrl.ticker_row, _scr_ctrl.ticker_col

         let _scr_ctrl.ticker_row = _scr_ctrl.ticker_row + 1
      end if
   end if

   initialize _calc_ctrl.cur_val to NULL
   initialize _calc_ctrl.prev_int to NULL

   let _calc_counter = _calc_counter + 1

   if _scr_ctrl.use_ticker and
      _scr_ctrl.ticker_row <= _scr_ctrl.NUM_VALS  then

      display _scr_ctrl.clear_number
            at  _scr_ctrl.wind_row, _scr_ctrl.wind_col
   end if

   return

end if

if c = "="  then   # do an equals op

  let _calc_ctrl.eq_flag = TRUE

  --
  -- if nothing has been input, just return
  --
  if _calc_counter = 1 then
     return
  end if

  --
  -- loop thru all values
  --
  let result = 0
  for n = 1 to _calc_counter - 1 step 2
```

```
    let result = result + do_op( _calc_ctrl.val[n],
                                 _calc_ctrl.val[n+1],
                                 _calc_ctrl.op[n] )

  end for

  display result using "<<<<<<<<.<<<<<<<<"
           at _scr_ctrl.wind_row, _scr_ctrl.wind_col
              attribute( reverse )

  let _calc_ctrl.last_result = result
  let e = _scr_ctrl.total_col + 8
  initialize _calc_ctrl.cur_val to NULL
  initialize c to NULL
  initialize _calc_ctrl.prev_int to NULL

  return

end if

if upshift( c ) = "C" then     -- clear the current number ⇐
    and number window

   display _scr_ctrl.clear_number
          at _scr_ctrl.wind_row, _scr_ctrl.wind_col

   initialize _calc_ctrl.cur_val to NULL

   return

end if

if upshift( c ) = "A"  then    -- clear ALL both list and ⇐
    val array

   call init_vals( )

   for n = 1 to _scr_ctrl.NUM_VALS
      let _calc_ctrl.val[n] = 0

   if _scr_ctrl.use_ticker   then
      display _scr_ctrl.clear_number, "    "
           at _scr_ctrl.ticker_row, _scr_ctrl.ticker_col
```

```
      let _scr_ctrl.ticker_row = _scr_ctrl.ticker_row + 1
   end if

   end for

   display _scr_ctrl.clear_number
        at _scr_ctrl.wind_row, _scr_ctrl.wind_col

   let _calc_counter = 1

   if _scr_ctrl.use_ticker then
      let _scr_ctrl.ticker_row = _scr_ctrl.TICK_ROW
   end if

   initialize _calc_ctrl.cur_val to NULL
   initialize c to NULL
   initialize _calc_ctrl.prev_int to NULL

   return

end if


call disp_number( _calc_ctrl.prev_int, 0 )  # clear previous ⇐
    int number
call disp_number( c, 1 )          # display this int number
let _calc_ctrl.prev_int = c

end function
```

The `disp_number` Function

The `disp_number` function is used to highlight a given number on the keypad. The object is to emulate the functionality of a desktop calculator. The function is passed two parameters: the number to be displayed and the attribute to use to display the number. The numeric value is received as a `smallint` in order to facilitate the mathematical operations required to determine where to display the value. A single character value is defined to hold the number to be displayed. The `display` statement does not require this, but placing the value in a single-character variable will help avoid any formatting problems when displaying the number. The `let` statement is used to convert the integer value passed to the function to the single character to be displayed.

If the numeric value passed to the function is NULL, which is possible given the logic of the application, the function will merely return at this point.

Since the keypad is arranged as a grid, a `case` statement is required to determine where the number passed to the function is to be displayed. Based on a number range, the `case` statement assigns x and y coordinates for display row and column, respectively.

The `y` value, or column coordinate, is determined based on a mathematical expression that reduces the number of lines of code. The numeric value is subtracted from the numeric value at the start of the keypad's row. The resulting value is then multiplied by the *spacing* value and added to the left column value for the keypad window. The `x` value, or row coordinate, is assigned by merely adding the starting row for the keypad box to the number row.

Once the `x` and `y` coordinates have been assigned, the attribute parameter is examined. If the attribute is set to zero for *normal* display attribute, then the number is displayed to the proper position in normal video attribute at the specified coordinates. If the attribute is set to one for the *reverse* video attribute, the number is displayed in reverse video at the specified coordinates.

```
function disp_number( n, attrib )
define n,attrib smallint
define disp_char char(1)
define x,y      smallint

if n is NULL then
   return
end if


let disp_char = n

   case

     when n >= 1 and n <= 3    let y = 10 + ( (n - 1) * SPACER )
                               let x = _scr_ctrl.START_ROW

     when n >= 4 and n <= 6    let y = 10 + ( (n - 4) * SPACER )
                               let x = _scr_ctrl.START_ROW + 1

     when n >= 7 and n <= 9    let y = 10 + ( (n - 7) * SPACER )
                               let x = _scr_ctrl.START_ROW + 2

     when n = 0                let y = 15
                               let x = _scr_ctrl.START_ROW + 3
```

```
   end case

   if attrib = 0 then
      display disp_char  at x,y
   end if

   if attrib = 1 then  -- bold
      display disp_char  at x,y
           attribute( reverse )
   end if

end function
```

The do_op Function

The do_op function performs the mathematical operation specified on the operands passed to the function as parameters. The operands passed to the function are type decimal with no specific precision, thus allowing the decimal point to float. The third parameter passed is a character representing the mathematical operation to perform. If either operand passed is a NULL, it is converted to a zero.

A case statement is used to evaluate the mathematical operation. All cases except division simply perform the operation requested and return the result. In the case of division, the division by zero error is trapped and an error message displayed, the operand converted to the value one and the result returned.

```
function do_op( val1, val2, op )
define val1, val2 decimal
define op char(1)
define ret_val decimal

if val1 IS NULL then
   let val1 = 0
end if
if val2 IS NULL then
   let val2 = 0
end if

case op

   when "+" return( val1 + val2 )

   when "-" return( val1 - val2 )
   when "*" return( val1 * val2 )
```

```
    when "/" if  val2 = 0 then
                 error "Attempt to divide by zero. "
                 let val2 = 1
             end if
             return( val1 / val2 )

    when "%" return( val1 mod val2 )

    when "E" return( val1 ** val2 )

end case

end function
```

The `init_vals` and `valid` Functions

The `init_vals` function does, as the name implies, initialization for the calculator application. This function populates the data control records for the application. Several key screen positions are defined, and then other values are defined relative to these screen positions.

The `valid` function is used to validate the character that has been typed at the keyboard. The `if/then` statement at the start of this function is an optimization that attempts to quickly evaluate the most commonly passed characters. If the number is a digit, or a C or A character, the function returns the value TRUE.

If that validation fails, the character is evaluated to determine whether it is an operator character. If it is any of these characters, the function returns TRUE immediately. If function execution reaches the end of the case statement and the character has not yet tested TRUE, it is assumed to be an invalid character and the value FALSE is returned.

```
function init_vals( )

-- keypad
let _scr_ctrl.START_ROW    = 10
let _scr_ctrl.START_COL    = 10

-- keypad window
let _scr_ctrl.wind_row = _scr_ctrl.START_ROW - 2
let _scr_ctrl.wind_col = _scr_ctrl.START_COL

let _scr_ctrl.PROMPT_LINE = 2
```

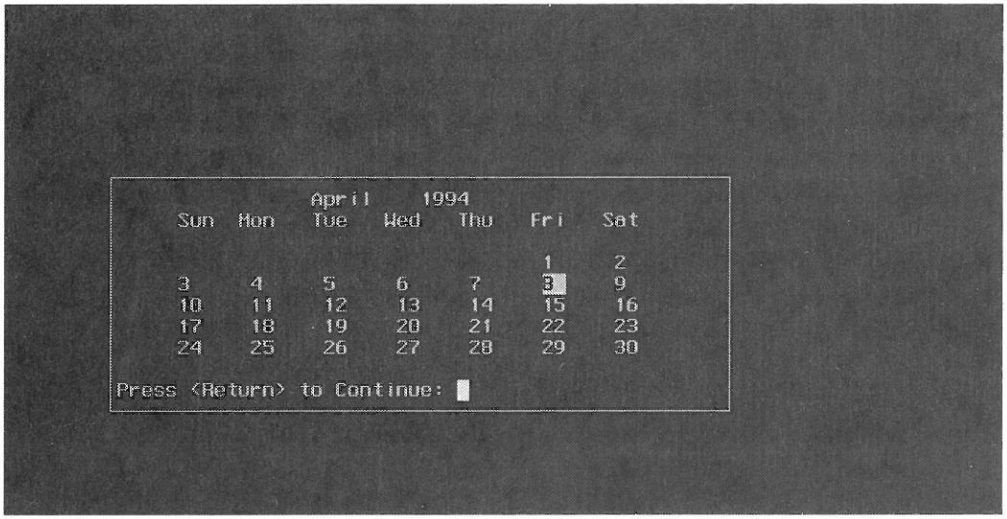
```
let _scr_ctrl.NUM_VALS    = 20

-- ticker box is relative to location of the keypad box
let _scr_ctrl.TICK_ROW    = _scr_ctrl.START_ROW - 6
let _scr_ctrl.ticker_row  = _scr_ctrl.TICK_ROW
let _scr_ctrl.ticker_col  = 40

let _scr_ctrl.total_row = 13
let _scr_ctrl.total_col = _scr_ctrl.ticker_col
let _scr_ctrl.clear_number = "                    "

let _calc_ctrl.eq_flag = FALSE

end function
------------------------------------------------------------

function valid( c )
define c char(1)

if ( c >= "0" and c <= "9" ) or
   ( upshift( c ) = "C" )    or
   ( upshift( c ) = "A" )     then
  return TRUE
end if

-- operators
case c
  when "=" return TRUE
  when "+" return TRUE
  when "-" return TRUE
  when "/" return TRUE
  when "*" return TRUE
  when "." return TRUE
  when "%" return TRUE
  when upshift(c) = "E" return TRUE
end case

return FALSE

end function
```

THE CALENDAR APPLICATION

The calendar application displays a calendar to the screen for the month and year passed, as shown in Figure 8-3. If the current day is in the month being displayed, it is displayed in reverse video.

As with the other applications, this application was designed to be merged or embedded into another application. There are no global variables; all variables are local to their function. The demonstration application shown below first accepts parameters for the month and year, and then passes these parameters to the `show_month` function for display to the screen. The user is then prompted to press return to clear the screen of the calendar. The implementation of this application is shown below.

Figure 8-3 The Calendar Application



The `main` Program Block and the `show_cal` Function

The `main` program block shown below simply calls the `show_cal` function twice with different sets of parameters. The first call passes the function NULL values, which will generate the default calendar; the current year and month will be displayed. The second call passes the month, year, screen row and screen column where the calendar will be displayed.

The `show_cal` function takes four parameters: the month to display, the year to display, the row at which to display the calendar and the column at which to display the calendar. The main program block for the sample program demonstrates several different calls to the `show_cal` function. The first call passes NULLs for all of the parameters, the result being that the default calendar is displayed: the calendar for the current month.

The next call displays the calendar for January of 1958 at row five, column five of the terminal screen. The remaining calls display the troublesome month of February for centennial years from 1500 to the year 2000; these calls were used to validate the leap year calculation in the `get_days` function explained below.

The `show_cal` function first validates the values passed into the function. If the values are NULL, the default values are substituted. If either the month or the year passed is NULL, the current year and month are substituted. If either the screen row or screen column passed are NULL, the default screen coordinates are used. Finally, the window for the calendar is opened and the `bld_month` function is called.

```
main

    call show_cal( "","","","" ) -- the default

    call show_cal( 1, 1995, 5,5 )

end main

------------------------------------------------------------
------------------------------------------------------------

function show_cal( pmonth, pyear, cal_row, cal_col )
define pmonth,
       pyear,
       cal_row,
       cal_col         smallint
define c char(1)

options prompt line last

if pmonth IS NULL or pyear IS NULL then
   let pmonth = month( today )
   let pyear  = year( today )
end if

if cal_row IS NULL or cal_col IS NULL then
    let cal_row = 10
    let cal_col = 10
end if

open window _cal1 at cal_row, cal_col
                with 10 rows, 50 columns
             attribute( border )
```

```
call bld_month( pmonth, pyear )

prompt "Press <Return> to Continue: " for char c

close window _call

end function
```

The bld_month Function

The bld_month function performs the bulk of the work for the calendar function. In the bld_month function, variables are created to hold the day and weekday, a header which will contain the mnemonics for the days of the week and a variable to hold the number of days in the month. The get_days function is passed the month to be displayed and returns the number of days in that month.

Next, the days_of_month array is set to blanks. This is done by performing a nested loop for each element of the two-dimensional array. The outer loop is from one to six, the number of weeks being built. The inner loop is from one to seven, the number of days in each week.

Next, the same nested loop is performed to load the days of the week into the days_of_month array. Within this loop, the lday variable is used as a counter to keep track of the current day of the month. The if/then statement below the counter is used to verify that the loop has not gone beyond the number of days in the month (stored in the daze variable). Following the end of the if/then statement, the weekday is found with the statement

```
let lweekday = weekday( mdy⇐
 ( pmonth, lday, pyear ) )
```

This statement makes use of two INFORMIX-4GL built-in functions. The mdy function will return the Informix format julian date for the month, day and year passed. This julian date is then passed to the weekday function, which will return a number between zero and six to indicate the weekday for the date passed.

The following line then assigns the correct element of the days_of_month variable with the value of the day counter stored in the lday variable. Since the days_of_month variable uses elements from 1 to 7 to store days, and the weekday function returns a number from 0 to 6 for the weekday, it is necessary to increment the weekday return value by one to access the correct element of the days_of_month variable.

```
let days_of_month[ n, (lweekday + 1)] = lday using "<<"
```

The next line evaluates the `lweekday` variable. If it is day number six, Saturday, it is necessary to move to the next week of the days_of_month array since the next day should be stored in the next week element of the `days_of_month` variable.

Next, the `lheader` string (which contains the name of the month and the year) is displayed, and on the following line the days of the week are displayed. On the following line, the same nested loop structure used previously is used to display the calendar to the screen. The outer loop counts the number of weeks from one to six and the inner loop counts the number of days.

Within this loop, the row is assigned to be the outer loop counter (the week) plus five, essentially displaying the calendar five lines from the top row of the window. The column position for the numbers is set by multiplying the inner loop count by six, effectively spacing the numbers six character positions apart. Next, the day of the month is examined. The statement

```
if days_of_month[ n, x ] = day( today )  and ⇐
 pmonth = month( today )
```

is used to determine if the current day, month and year being displayed in the loop is today's day and month. If this statement tests TRUE, the day being displayed is displayed in reverse video, thus highlighting the current day. If it tests FALSE, the day is displayed in normal video.

```
function bld_month( pmonth, pyear )
-- build strings with weeks in month
define pmonth, pyear smallint
define days_of_month array[6, 7] of char(2)
define lday integer
define x,y,n, daze, prow, pcol smallint
define lweekday smallint
define lheader char(40)
define ldate  char(12)
define daystr char(2)

let daze  = get_days( pmonth, pyear )

# set strings to null
for n = 1 to 6
  for x = 1 to 7
   let days_of_month[n][x] = "  "
  end for
end for
```

```
let lday = 0

--
-- fill the date array
--
for n = 1 to 6

 for x = 1 to 7

   let lday = lday + 1

   if lday > daze  then
      exit for
   end if

   let lweekday = weekday( mdy( pmonth, lday, pyear ) )
   let days_of_month[n, (lweekday + 1)] = lday using "<<"

   -- if on a Sat, need to move to the next week
   if lweekday = 6  then
      exit for
   end if

 end for

end for

--
-- display the calendar
--
let lheader = get_month( pmonth ), "     ", pyear using "<<<<"
display lheader
       at 1,17
display "Sun  Mon   Tue   Wed   Thu   Fri   Sat"
       at 2,6

for n = 1 to 6

  let prow = n + 3

  for x = 1 to 7
     let pcol   = x * 6
```

```
      if days_of_month[ n, x ] = day( today )  and
         pmonth = month( today )                and
         pyear = year( today )  then          # it's our current
                                              # month/day/year

            display days_of_month[ n, x ] at prow, pcol
                    attribute( reverse )
      else
            display days_of_month[ n, x ] at prow, pcol
      end if

   end for

end for

end function

---------------------------------

function get_month( pmonth )
define pmonth smallint

case pmonth
   when 1  return( "January" )
   when 2  return( "February" )
   when 3  return( "March" )
   when 4  return( "April" )
   when 5  return( "May" )
   when 6  return( "June" )
   when 7  return( "July" )
   when 8  return( "August" )
   when 9  return( "September" )
   when 10 return( "October" )
   when 11 return( "November" )
   when 12 return( "December" )
end case

end function
```

The `get_month` and `get_days` Functions

The `get_month` and `get_days` functions perform simple operations that are used in the calendar application. The `get_month` function takes a month number and

maps the number to the character string name of the month number via a `case` statement. This character string is then returned to the calling function.

The `get_days` function takes a month number and returns the number of days in that month. It must make a simple calculation in the month of February to determine whether or not the year is a leap year and if the month has 28 or 29 days. Otherwise, it makes use of two `if/then` statements to return the correct number of days to the calling function.

```
function get_days( pmonth )
define pmonth smallint


 -- handle those blasted leap years
 if pmonth = 2  then

   if NOT ( pyear mod 100 )  then # - centennial year
      if NOT ( pyear mod 400 )  then # - and divisible by 4,
                                     # therefore it's a leap year
         return 29
       end if
   else                  -- not a centennial yr
      if NOT ( pyear mod 4 )  then # - it's a leap year
         return 29
      end if
   end if

   -- else, not a leap year
   return 28

 end if

 if pmonth = 1  or
    pmonth = 3  or
    pmonth = 5  or
    pmonth = 7  or
    pmonth = 8  or
    pmonth = 10 or
    pmonth = 12    then

      return 31
 end if

 if pmonth = 4  or
```

```
    pmonth = 6  or
    pmonth = 9  or
    pmonth = 11   then

     return 30

 end if

end function
```

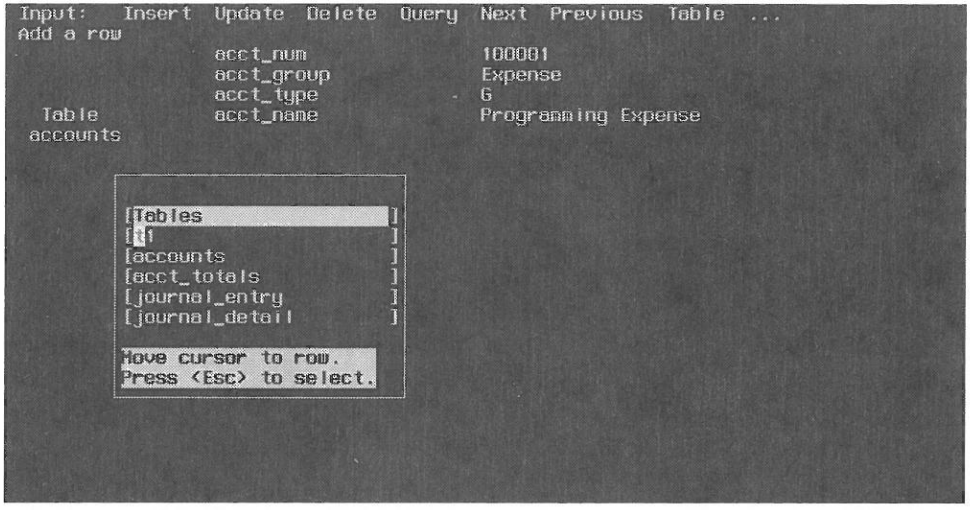
Chapter 9

Table Browser Application

The `gen_tab` application is a generalized table-browsing application written entirely in INFORMIX-4GL. It takes the name of a database and a table as parameters and then displays a screen form and menu to allow the contents of the table to be browsed and modified, as shown in Figure 9-1. It provides a ring menu with a variety of options for reviewing and modifying existing table records, and adding new records.

Optionally, if no table parameter is passed to the application, a pick list of tables will be displayed allowing the user to choose a table. If the table has more than 20 columns, the user can move back and forth between the pages of columns using control keys. Full query-by-example is also supported.

Figure 9-1 Table Browser Application Screen



The table browser is designed around two global data structures. These data structures carry information about the current table being browsed such as the names of the columns, the data types, length and the current value of the columns. The data structures are arrays dimensioned to 60 elements. Each element corresponds to a table column, so if a table has more than 60 columns, the table browser will only browse 60 of them.

This application takes liberal advantage of INFORMIX-4GL type conversion. All data values are treated as strings; it is up to the database engine to perform conversion

when the data is inserted into the database, and it is up to INFORMIX-4GL to convert the table column data types to strings when they are retrieved.

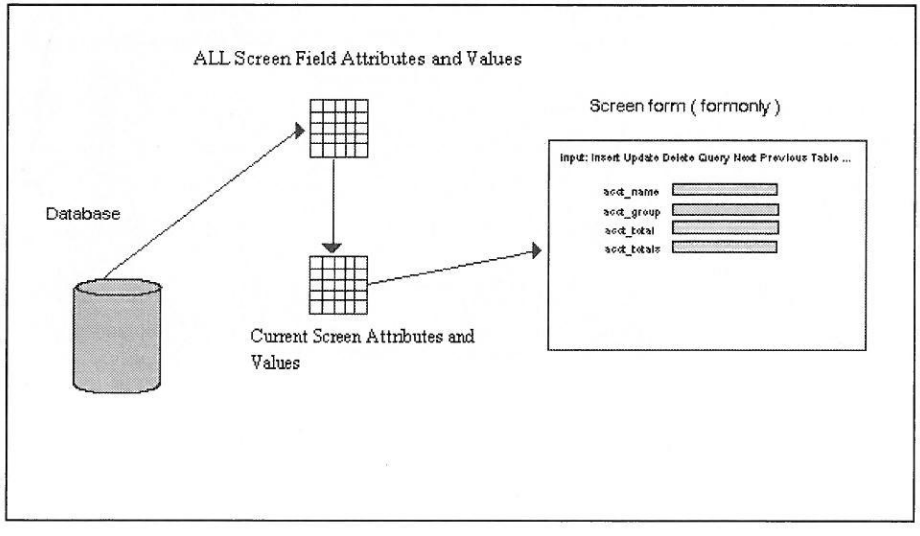
The screen form is a formonly screen; it is not bound to any database or database tables. This allows it to be used independent of any particular database.

The application builds queries by scanning the list of table columns and appending the appropriate column names to a string. The string is then used to build a query for a scroll cursor. The scroll cursor is then used to allow the user to step forward and back in the current select list.

Updates and deletes are performed using `rowid` access. While not always a recommended method of accessing data rows, in this case, the rowid access provides a convenient means of accessing a specific row in a table where the composition of the primary key is unknown.

Multiple screens are supported, allowing tables of up to 60 columns to be browsed using this application. The Control-F and Control-B keys are used to move forward and back a screen, respectively. Multiple screen support is implemented by maintaining two internal arrays. One array corresponds to the current screen and the other corresponds to the entire data record. Much of the complex code in the application is devoted to keeping these arrays synchronized (see Figure 9-2).

Figure 9-2 Table Browser Array Management



THE PROGRAM MAIN

The main program block processes command line arguments, opens the database and then passes control to the main program loop. If a database name is not passed to the

program, it cannot operate. The program will display an error message and then terminate. If the program has successfully been passed a database name and a table name, it then displays a message indicating that it is initializing.

If no table name has been passed, the `zoom1` function will be called and passed the name of the `systables` system catalog table. Since all system catalog tables in the Informix database have a table id (`tabid`) less than 99, the `where` clause criteria passed to the `zoom1` function is "`tabid > 99`". This will produce a listing of tables in the current database that are not system catalog tables. When the user moves to the table they wish to browse and presses the ACCEPT key, the table name is returned and inserted in the `tab_name` variable.

The function `gen_tab_input`, which executes the main menu for the application, is called with the name of the table for which to perform input. This single function, if passed a table name for a table in the current database, will perform all of the functions required to browse and update that table. It then returns a code indicating how the user exited the function. This modularity allows the table browser application to be easily inserted and merged into an existing application.

All of this is performed within a `while` loop, which will loop until the `ret_code` equals "E", indicating that the user selected the "exit" option from the menu. If the user has not selected the "exit" option, a pick list is displayed, allowing them to choose another table in the database. If the user chooses another table, the `gen_tab_input` function is called again with the name of the new table.

```
main
define tab_name  char(20)
define db_name   char(20)
define num_flds  smallint
define msg       char(60)
define prog_name char(10)
define ret_code  char(1)
initialize msg      to null
initialize db_name  to NULL
initialize tab_name to NULL

if num_args() = 1 then
   let db_name = arg_val( 1 )
end if

if num_args() = 2 then
   let db_name  = arg_val( 1 )
   let tab_name = arg_val( 2 )
end if
```

```
--
-- Open the database
--
if dbname IS NULL then
   error "Database name is required."
   sleep 2
   exit program
end if

clear screen
let msg = "** Opening Database ",db_name clipped, " ** "
let msg = center( msg, 80 )
display msg clipped at 10,1
database db_name


if tab_name IS NULL then  -- display a pick list for table name
   call zoom1( "systables", "tabname", "tabid > 99", "Tables" )
        returning tab_name
end if
--
-- Enter input loop
--

let ret_code = "C"
while ( ret_code != "E" )

   if tab_name IS NULL then

      call zoom1( "systables", "tabname", "tabid > 99", "Tables" )
          returning tab_name

      if tab_name IS NULL or  upshift( tab_name ) = "Q" then
         exit while
      end if

   end if

   let msg = center( "Initializing ... ", 80 )
   display "" at 10,1
   display msg clipped at 10,1
   let ret_code =  gen_tab_input( tab_name )
```

```
   initialize tab_name to NULL

end while

end main
```

THE GLOBALS SECTION

A number of global variables are used to track the function of the gen_tab application. In order to enhance modularity and make it easier to integrate this application into existing code, the number of global variables is kept to a bare minimum. Only three records and seven simple variables are used to control the application.

The fld_arrs record is an array of records that track information about the database fields being manipulated. The field_name corresponds to the colname column in the Informix syscolumns table. The field_label record element is an optional label to be displayed on the terminal screen. This is loaded from the gen_cols table in the load_fields function. If the gen_cols table does not exist in the current database, this data is not loaded and the default field labels, the column names from the syscolumns table system catalog table, are used.

```
globals

--
--
--
define flds_arr array[20] of
  record
    field_name    char(20),
    coltype       integer,
    collength     integer,
    val           char(40)
  end record

define mstr_flds_arr array[60] of -- add multiple screens support
  record
    field_name    char(20),
    coltype       integer,
    collength     integer,
    val           char(40)
  end record

define prowid  integer
define MAXFIELDS   smallint
```

```
   initialize tab_name to NULL

end while

end main
```

THE GLOBALS SECTION

A number of global variables are used to track the function of the gen_tab application. In order to enhance modularity and make it easier to integrate this application into existing code, the number of global variables is kept to a bare minimum. Only three records and seven simple variables are used to control the application.

The fld_arrs record is an array of records that track information about the database fields being manipulated. The field_name corresponds to the colname column in the Informix syscolumns table. The field_label record element is an optional label to be displayed on the terminal screen. This is loaded from the gen_cols table in the load_fields function. If the gen_cols table does not exist in the current database, this data is not loaded and the default field labels, the column names from the syscolumns table system catalog table, are used.

```
globals

--
--
--
define flds_arr array[20] of
  record
    field_name    char(20),
    coltype       integer,
    collength     integer,
    val           char(40)
  end record

define mstr_flds_arr array[60] of -- add multiple screens support
  record
    field_name    char(20),
    coltype       integer,
    collength     integer,
    val           char(40)
  end record

define prowid  integer
define MAXFIELDS   smallint
```

```
define TOTALFIELDS smallint
define CHARCOL     smallint
define NEXT, PREV  smallint


-- used to manage the screen field list
define scr_control record
  disp_flds,         -- the number of fields currently displayed
  curr_fld_offset,   -- current offset into num_flds loaded
  curr_scr_start,    -- start field number of current screen
  curr_scr_end,      -- ending field number of current screen
  total_num_flds     -- total number of fields in the screen form
    smallint
end record

define debug smallint

end globals
```

THE gen_tab_input FUNCTION

The gen_tab_input function is the top-level function for this application. It is the main entry point for the table browser program. It takes a single parameter, the name of a database table. It defines a variable to hold the query string and a single-character return code. It then performs initialization on some of the variables that are used as *constants* in the application.

Next, a load_fields function is called with the name of the table. The load_fields function will populate the global array of records, which track information about the fields. This function will load the names of the table's columns and any field labels that may exist in the database into the global record array. Field labels are stored in a database table named gen_cols. In the load_fields function, which will be reviewed later in this chapter, the existence of this table is verified; if the table does not exist, there is no attempt to load the field labels. If the table does exist, the field labels are retrieved into the global record array. The load_fields function returns the total number of fields retrieved. This number is stored in the scr_control record and is used throughout the application to maintain screen field input control.

The next few lines of code assign values to the scr_control record. The screen control record (scr_control) contains information on the current state of the screen. It essentially groups together a number of variables used to maintain the sanity of the screen and the program arrays that contain column attributes and values. The current offset into the program array (mstr_flds_arr), the current screen start, the current screen end and the number of fields currently being displayed are all stored in this record.

It is essential that the scr_control record be set up correctly, or the screen cannot be managed by the application. First, the curr_fld_offset element is assigned the value of the one, indicating the current field list displayed on the screen begins with the first field from the table. This is what would be expected for the first page of screen input.

Next, the total number of fields in the screen control record is examined. This is the value that was returned by the load_fields function (scr_control.total_num_flds). If the total number of fields is less than the maximum number of fields that can be displayed on the screen at once (MAXFIELDS), this is an indication that we have enough data to display on a single screen—we do not have multiple pages of data to display. We can therefore set the value of the disp_flds record element to the total number of fields element, the value returned by the load_fields function.

```
if scr_control.total_num_flds < MAXFIELDS  then
   let scr_control.disp_flds = scr_control.total_num_flds
else                              -- we have multiple pages
   let scr_control.disp_flds = MAXFIELDS
end if
```

If the total number of fields value is greater than the number of fields that can be displayed on a single screen, the disp_flds record element is set to the maximum number of fields that can be displayed on a single screen. This is an indication that we have multiple pages of data to display, but the most we can display on a single page is the value that we store in the disp_flds element. (It should be clear that to alter the number of fields to be displayed on a single screen, the MAXFIELDS parameter could simply be changed.)

Next, the init_mstr_flds functions are called. These functions will initialize the values in the mstr_flds_arr and the flds_arr statement, respectively. After this function has been called, the scr_page function is called to place the screen on page one. This function will set the scr_control record to the correct settings for page one and will load the correct field labels for the screen.

At this point , the display_form function is read to be called. This function is passed parameters for the table and the number of fields to display. The display_form function will open and display the form for the table up to the number of fields specified in the disp_flds parameter.

Once the form is displayed, the menu can then be displayed. The gen_tab application menu has options for *insert*, *update*, *delete*, *query*, *next*, *previous*, *table* and *exit* to move forward one page of fields, to move back one page of fields, to change the current table or to exit the application. These options are explained below.

MENU OPTIONS

For the **insert**, **update** and **delete** operations, single functions are called to perform the operation. For the **query** function, a series of functions are called. First, the `gen_query` function is called to execute a `construct` statement on the screen fields and return the appropriate SQL `where` clause with filter criteria. Next, the `scr_page` function is called to bring the screen form back to the first page. The `gen_fetch` function is then called to retrieve the first page of data from the query. Once the data is retrieved, it is displayed on the screen using the `gen_disp` function.

The **next** and **previous** options first make a call to the `gen_fetch` function to retrieve the data and then call the `gen_disp` function to display the data that has been retrieved.

The **table** option allows a new table to be chosen. Since the `gen_tab` function, in which this menu resides, expects to act on a single table, the easiest way to allow the application to browse a new table would be to simply call this function again. Rather than incur the overhead of doing this recursively, table option simply exits with a return code of T. This tells the upper-level function that called `gen_tab_input`, the main function block, that a new table should be chosen. (If the main program block were to receive a return code of E it would not prompt for a new table but would simply exit the program.)

The **forward page** and **back page** options move the screen forward and back a page, assuming there is more than one screen of table columns to see. These options call a single function that handles both forward and back page functions. The function `move_flds_page` is passed parameters for table name and an integer value of one, which indicates a forward move. This will cause the `move_flds_page` function to prepare the `mstr_flds_arr` and `scrn_flds_arr` record arrays for the forward move. If the function is passed an integer negative one, it will prepare the `mstr_flds_arr` and the `flds_arr` arrays for moving back one page. This function will indicate whether or not the page is available and display an error message if the page is not.

If the **exit** option is chosen from the menu, the `ret_code` variable is set to E to indicate that the user has chosen to exit the menu. Following the end of the menu block, the `ret_code` is returned to the calling function. This statement represents the end of the `gen_tab_input` function.

```
function gen_tab_input( tab )
define tab char(20)
define qstr char(500)
define ret_code char(1)
-- --
-- --these must be initialized
-- --
```

```
let MAXFIELDS     = 20
let TOTALFIELDS   = 60
let CHARCOL       = 0
let NEXT          = 1
let PREV          = -1


-- initialize these fields
call init_mstr_flds( MAXFIELDS )
call init_flds_arr( TOTALFIELDS )

call load_fields( tab )
      returning scr_control.total_num_flds

let scr_control.curr_fld_offset = 1
if scr_control.total_num_flds < MAXFIELDS  then
   let scr_control.disp_flds = scr_control.total_num_flds
else                              -- we have multiple pages
   let scr_control.disp_flds = MAXFIELDS
end if

call load_fields( tab )
      returning scr_control.total_num_flds
-- set screen control record elements to page 1
call scr_page( tab, 1 )
call display_form( tab, scr_control.disp_flds )

menu "Input"

    command "Insert"   "Add a row"

            call gen_add( tab, scr_control.total_num_flds )

    command "Update"   "Update a row"

           call gen_update( tab, scr_control.disp_flds )


    command "Delete"   "Delete a row"

           call gen_delete( tab )
```

```
    command "Query"    "Query a row"

            call gen_query( tab, scr_control.total_num_flds )
                 returning qstr
            call gen_open( qstr )
            call scr_page( tab, 1 )            -- snap back to ⇐
     first page
            call gen_fetch( NEXT )
            call gen_disp( tab, scr_control.disp_flds )

    command "Next"     "Next row"

            call gen_fetch( NEXT )
            call gen_disp( tab, scr_control.disp_flds )

    command "Previous" "Previous row"

            call gen_fetch( PREV )
            call gen_disp( tab, scr_control.disp_flds )


    command "Table" "Choose New Table"

            let ret_code = "T"
            exit menu

    command "Forward_page"  "Show Next Page of Fields "

            call move_flds_page( tab, +1 )

     command  "Back_page"  "Show Previous Page of Fields "

            call move_flds_page( tab, -1 )

    command "Exit"

            let ret_code = "E"
            exit menu
end menu

return ret_code

end function
```

The `display_form` Function

The `display_form` function shown below is used to display the screen field labels to the screen. The screen field labels will be either the name of a specific column or a field label name from the `gen_cols` table. The function receives as parameters the table name and the number of fields for the table.

The screen form for the application is shown below. This screen form creates a set of field labels, labeled `f#` on the screen form, and input fields, labeled `i#` on the screen form. All fields have the default data type of character, since the `gen_tab` application will treat all data as the character data type. Note that the form is a `formonly` form and that the fields are a series of independent fields, not a screen array.

```
database formonly

screen
{
                [f1                ] [i1 ]
                [f2                ] [i2                ]
                [f3                ] [i3                ]
  Table         [f4                ] [i4                ]
[t1         ]   [f5                ] [i5                ]
                [f6                ] [i6                ]
                [f7                ] [i7                ]
                [f8                ] [i8                ]
                [f9                ] [i9                ]
                [f10               ] [i10               ]
                [f11               ] [i11               ]
                [f12               ] [i12               ]
                [f13               ] [i13               ]
                [f14               ] [i14               ]
                [f15               ] [i15               ]
                [f16               ] [i16               ]
                [f17               ] [i17               ]
                [f18               ] [i18               ]
                [f19               ] [i19               ]
                [f20               ] [i20               ]
}
end
attributes
f1 = formonly.f1;
f2 = formonly.f2;
f3 = formonly.f3;
f4 = formonly.f4;
```

```
f6 = formonly.f6;
f7 = formonly.f7;
f8 = formonly.f8;
f9 = formonly.f9;
f10 = formonly.f10;
f5 = formonly.f5;
f11 = formonly.f11;
f12 = formonly.f12;
f13 = formonly.f13;
f14 = formonly.f14;
f15 = formonly.f15;
f16 = formonly.f16;
f17 = formonly.f17;
f18 = formonly.f18;
f19 = formonly.f19;
f20 = formonly.f20;

t1  = formonly.table_name;

i1 = formonly.i1;
i2 = formonly.i2;
i3 = formonly.i3;
i4 = formonly.i4;
i5 = formonly.i5;
i6 = formonly.i6;
i7 = formonly.i7;
i8 = formonly.i8;
i9 = formonly.i9;
i10 = formonly.i10;
i11 = formonly.i11;
i12 = formonly.i12;
i13 = formonly.i13;
i14 = formonly.i14;
i15 = formonly.i15;
i16 = formonly.i16;
i17 = formonly.i17;
i18 = formonly.i18;
i19 = formonly.i19;
i20 = formonly.i20;

instructions

  delimiters "  "


end
```

The `display_form` function does not display a screen form in the traditional sense. The screen form for the `gen_tab` application is a set of fields; there are no text labels in those fields. All field labels for the input fields are displayed from the `flds_arr` record elements at runtime.

The function begins by opening the screen form and displaying it to the screen. The name of the table is then displayed on the `tab_name` field on the screen form, and each field name element of the `flds_arr` array is then displayed to its corresponding screen form field. A `for` loop is used to display the elements to the screen up to the value of `num_fields`, the number of fields currently displayed on the screen.

When the function has displayed all the fields on the screen, it simply returns to the calling function. The `case` statement is necessary here because the field names in a `display` statement must be character constants; they cannot be character expressions.

```
function display_form( tab_name, num_fields )
define tab_name   char(20)
define num_fields smallint
define n smallint
define hdr char(60)

open form f1 from "gen_tab"
display form f1
clear form

display tab_name to table_name;

-- --
-- --display field names
-- --
for n = 1 to num_fields

   case n
        when 1  display flds_arr[n].field_name to f1
        when 2  display flds_arr[n].field_name to f2
        when 3  display flds_arr[n].field_name to f3
        when 4  display flds_arr[n].field_name to f4
        when 5  display flds_arr[n].field_name to f5
        when 6  display flds_arr[n].field_name to f6
        when 7  display flds_arr[n].field_name to f7
        when 8  display flds_arr[n].field_name to f8
        when 9  display flds_arr[n].field_name to f9
        when 10 display flds_arr[n].field_name to f10
        when 11 display flds_arr[n].field_name to f11
```

```
        when 12 display flds_arr[n].field_name to f12
        when 13 display flds_arr[n].field_name to f13
        when 14 display flds_arr[n].field_name to f14
        when 15 display flds_arr[n].field_name to f15
        when 16 display flds_arr[n].field_name to f16
        when 17 display flds_arr[n].field_name to f17
        when 18 display flds_arr[n].field_name to f18
        when 19 display flds_arr[n].field_name to f19
        when 20 display flds_arr[n].field_name to f20
    end case


end for

end function
```

The `load_fields` Function

The `load_fields` function retrieves the names of the table columns and the corresponding field label names from the database and stores this information in the `mstr_flds_arr` array of records.

This function will optionally load the field name labels and the corresponding column entries for the table from the `gen_cols` table if it exists. It is necessary to allow for the possibility that the `gen_cols` table does not exist in the database. The `get_count` function is used to first determine if there is a table in the database named `gen_cols` and, if it exists, if there is an entry for the current table in the `gen_cols` table. If both of these conditions are true, a query is built that retrieves both the `colname` from `syscolumns` table and the `collabel` from the `gen_cols` table with the following query.

```
let qstr = " select c.colname, g.collabel,
           coltype, collength ",
                " from systables t, syscolumns c,
                 gen_cols g ",
                " where c.tabid = g.tabid and ",
                " c.colno = g.colno and ",
                " t.tabid = c.tabid  and ",
                " t.tabname = ", "'", tab clipped,
                "'"
```

This query performs a three-way join between the `systables`, `syscolumns` and `gen_cols` tables. It does not perform an outer join on the `gen_cols` table; the assumption is that if the `gen_cols` table exists and there are entries for the current

table, there are in fact entries for **all** columns in the current table. If this is not the case, the `gen_tab_input` function will not perform correctly.

If there is no `gen_cols` table, or there is a `gen_cols` table but there are no entries for the current table, a query is performed that simply joins the `syscolumns` table with the `systables` table.

Once the structure of the query has been determined, the query is prepared and a cursor is declared for the prepared statement. The `foreach` statement is then used to retrieve the data into the array elements of the `mstr_flds_arr` array. A counter is kept while the `foreach` loop is executing. If the counter becomes greater than the maximum number of elements allowed, an error message is displayed indicating that the table has too many columns, and the `foreach` loop will be exited. The program execution will continue at this point; the user will simply not be able to view or modify the columns that have not been loaded. When complete, the function returns the number of columns that have been loaded into the `mstr_flds_arr` program array.

```
function load_fields( tab )
define tab  char(20)
define qstr char(500)
define n    smallint
define tab1 char(30)
define col1 char(20)
define join1 char(60)

-- if there is a gen_cols table on the system, we'll use it for
-- the column names
if get_count( "systables", "tabname", "gen_cols" )  and
   get_count( "gen_cols", "tabname", tab )            then

 let qstr = " select c.colname, g.collabel, coltype, collength ",
               " from systables t, syscolumns c, gen_cols g ",
               " where c.tabid = g.tabid and ",
               " c.colno = g.colno and ",
               " t.tabid = c.tabid  and ",
               " t.tabname = ", "'", tab clipped, "'"
else

   let qstr = " select colname, colname, coltype, collength ",
              " from systables t, syscolumns c ",
              " where c.tabid = t.tabid  and ",
              " t.tabname = ","'", tab clipped, "'"

end if
```

```
prepare _st1 from qstr
declare _c1 cursor for _st1

--
-- load the fields to be displayed
--
let n = 1
foreach _c1 into mstr_flds_arr[n].field_name,
                 mstr_flds_arr[n].field_label,
                 mstr_flds_arr[n].coltype,
                 mstr_flds_arr[n].collength
let n = n + 1
if n > TOTALFIELDS  then
  error "Too many fields in table ",tab
  exit foreach
end if

end foreach

--
-- return the number of fields loaded
--
return ( n  - 1 )

end function
```

The gen_add Function

The gen_add function is called when the user selects the **insert** option off the menu displayed in the gen_tab_input function. This function takes the name of the table and the number of fields as parameters. Initially, the scr_control record is checked to determine if the screen is on the first page; if it is not, the program is moved to the first page by calling the scr_page function.

Next, the gen_input function is called to allow the user to enter data on the screen. This function is called with four arguments: the name of the table being input, the number of fields to input, a flag indicating whether or not to validate the data being entered, and a character string indicating whether or not data is being entered as part of an **insert** or **add** operation, or **update** or **modify** operation.

In the next step, the first part of the SQL insert statement is entered and a for loop is started for each field in the table. Before data is entered in the gen_input function, the mstr_flds_arr and flds_arr have been initialized to NULL; if no data has been entered, the value of the field should still be NULL. If a NULL value element is found while scanning the mstr_flds_arr, a continue for statement is

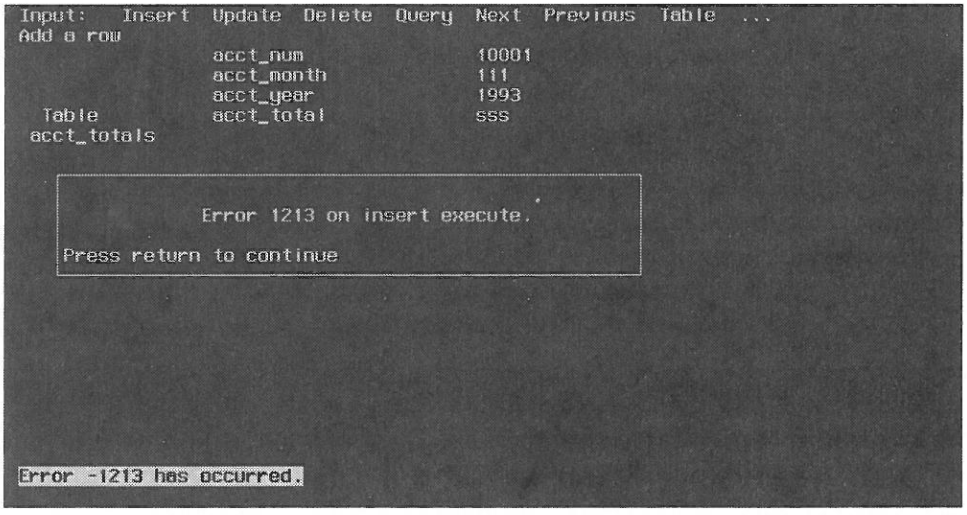
executed to continue the `for` loop. If the value (`val`) element of the `mstr_flds_arr` is NOT NULL, it is appended onto the SQL statement being built for the SQL insert operation. Once all of the field names have been concatenated onto the SQL **insert** statement, the statement now appears as a string similar to

```
insert into customer (first_name, last_name, city )
```

The last part of the `insert` statement must now be built. This is done through a similar `for` loop, which will now add the values to the `insert` statement. As the values are added, they are surrounded with single quotes. This will force the Informix database engine to evaluate the values first as strings, and then convert them into the appropriate data type for their corresponding column. This allows the application to avoid the subject of data typing when executing the `insert` statement.

Both loops use a counter variable named `flds_in_list`. If, at the end of the second loop, this counter variable is set to zero, it can be asserted that there are no values to insert into the database. In this case, the function displays a message indicating that no values were inserted (as shown in Figure 9-3) and returns to the calling function.

Figure 9-3 Table Browser Insert Error Message



If, however, the value of the counter variable is positive, the insert operation is initiated. First, a `whenever error continue` statement is executed to trap errors. The SQL `insert` statement is prepared and the results evaluated for an error condition. If an error has occurred, an error message is displayed and the function returns. If no error has occurred, the statement is executed using the `execute` statement and the result is tested for errors. If there are no errors after the insert operation, the function merely

returns to the calling function. If an error has occurred, an error message is displayed (as shown below), and the function returns to the calling function; program execution will continue.

```
function gen_add( tab, num_flds )
define tab char(20)
define num_flds,n smallint
define sql_stmt char(2000)
define msg char(70)
define flds_in_list smallint

if scr_control.curr_scr_start != 1 then -- not on page 1
   call scr_page( tab, 1 )
end if

call gen_input( tab, num_flds, TRUE, "ADD" )

let sql_stmt = " insert into ", tab clipped, "("

-- load the field names

let flds_in_list = 0
for n = 1 to num_flds

    if  mstr_flds_arr[n].val IS NULL  then
        continue for
    end if

    let flds_in_list = flds_in_list + 1
    if flds_in_list > 1 then
       let sql_stmt = sql_stmt clipped, ","
    end if
    let sql_stmt = sql_stmt clipped,
                   mstr_flds_arr[n].field_name clipped

end for

let sql_stmt = sql_stmt clipped, " ) values ( "

-- load the field values
let flds_in_list = 0
for n = 1 to num_flds
```

```
   if mstr_flds_arr[n].val IS NULL  then
      continue for
   end if

   let flds_in_list = flds_in_list + 1
  if flds_in_list > 1  then
      let sql_stmt = sql_stmt clipped, ","
   end if
   let sql_stmt = sql_stmt clipped, "'", mstr_flds_arr[n].⇐
    val clipped, "'"

end for
let sql_stmt = sql_stmt clipped, " ) "

if flds_in_list = 0  then  -- nothing to add
   error "Nothing inserted."
   return
end if

whenever error continue

prepare _istmt1 from sql_stmt
if sqlca.sqlcode < 0  then
   let msg = sqlca.sqlcode using "Error <<<< on insert prepare."
   error "Error ",sqlca.sqlcode, " has occurred."
   call err_msg( msg )
end if

execute _istmt1
if sqlca.sqlcode < 0  then
   let msg = sqlca.sqlcode using "Error <<<< on insert execute."
   error "Error ",sqlca.sqlcode, " has occurred."
   call err_msg( msg )
end if
whenever error stop


end function
```

The `gen_update` Function

The `gen_update` function operates in a manner similar to that of the `gen_add` function. Since an update operation is being performed, it would be desirable to avoid updating any columns that have not been touched. In order to do this, the application must

store the field values before the input operation, and then compare those values against the values existing after the input operation.

Before the input operation is performed, a `for` loop is executed to load the current values of the `mstr_flds_arr` into an array of character strings. After the input operation, this array will be used to determine whether or not the contents of a field have been changed during the input operation.

Next, the application forces the screen back to the first page before calling the `gen_input` function. When the `gen_input` function is called, it is called with a parameter which indicates that an update operation is being performed. Upon return from the `gen_input` function, the `mstr_flds_arr` array is scanned to determine first if the field is non-null, and second, whether or not the contents of the field have changed. If both of these conditions are satisfied, the field name and the updated value of the field are concatenated onto the string that is being loaded with the `update` statement. A `fields_list` counter is kept to determine how many fields have been added to the `update` statement. If, at the end of the `for` loop, this counter is zero, this is an indication that there are no fields to be updated, and the function will simply display a message indicating that nothing was updated and return to the calling function.

Finally, the error-checking mode is switched to `whenever error continue` to allow the error checking to be localized. The SQL update statement is prepared using the `rowid` of the current row as a means of accessing that specific row. While there are some problems associated with the use of `rowid` access, it can provide an easy and fast method of accessing the current row in a table when the primary key is not known, as this example demonstrates.

After the SQL `update` statement is prepared, the `sqlca` record is examined for errors. If an error has occurred, an error message is displayed and program execution will continue.

If the `prepare` of the `update` statement succeeded, it is executed and the results of that execution are checked for errors. If an error has occurred, an error message is displayed on the screen. If no errors have occurred, the function will quietly return to the calling function.

```
function gen_update( tab, num_flds )
define tab char(20)
define num_flds,n, flds_in_list smallint
define sql_stmt char(1000)
define msg  char(60)
define prev_flds_arr array[60] of record
           val char(40)
 end record

--
```

```
-- store the current field values
-- will examine these after the input operation
--
for n = 1 to scr_control.total_num_flds

   if mstr_flds_arr[n].val IS NULL then
      let prev_flds_arr[n].val = " "
   else
      let prev_flds_arr[n].val = mstr_flds_arr[n].val
   end if

end for

let sql_stmt = "update ", tab clipped, " set "

while  scr_control.curr_scr_start != 1     -- not on page 1
    call move_flds_page( tab, -1 )
end while
call gen_input( tab, num_flds, TRUE, "UPDATE" )

--
-- compare current values with previous values
-- update only modified fields
--
let flds_in_list = 0
for n = 1 to scr_control.total_num_flds + 1

  if  mstr_flds_arr[n].val IS NOT NULL  and
      mstr_flds_arr[n].val != prev_flds_arr[n].val then
      -- field touched

           let flds_in_list = flds_in_list + 1
         if flds_in_list > 1 then
            let sql_stmt = sql_stmt clipped, ","
          end if
         let sql_stmt = sql_stmt clipped, " ",
                        mstr_flds_arr[n].field_name clipped,
                        " = ", "'", mstr_flds_arr[n].⇐
    val, "'"  clipped
     end if

end for

if flds_in_list = 0  then    -- nothing was updated
  error "Nothing updated."
  return
```

```
end if

--
-- there are values to update
--
let sql_stmt = sql_stmt clipped,
           " where rowid = ",prowid


--
-- perform the udpate
--
whenever error continue
prepare _ustmt1 from sql_stmt
if sqlca.sqlcode < 0 then
   let msg = sqlca.sqlcode using "Error <<< on update prepare."
   call err_msg( msg )
end if
execute _ustmt1
if sqlca.sqlcode < 0 then
   let msg = sqlca.sqlcode using "Error <<< on update execute."
   call err_msg( msg )
end if
whenever error stop

end function
```

The `gen_delete` Function

The `gen_delete` function is used to delete the current row pointed to by the `rowid` stored in the global variable `prowid`. The function begins by calling the `prompt_msg` function to display a message prompt to the screen. The message asks the user if they want to delete the record. The user can respond to the prompt with a Y for yes, delete the current row, or an N for no, do not delete the current row. The single-character response is shifted to uppercase with the `upshift` function and evaluated against the uppercase letter Y. If the user has typed *any* character other than Y, the delete operation is canceled. The `err_msg` function is called to display the error message (as shown in Figure 9-4) and prompts the user to press the return key to continue.

If the user has responded with the letter Y, the delete operation is started (see Figure 9-5). First, the query string is built using the global `prowid` variable. The `whenever error continue` mode of error trapping is started. The SQL `delete` statement is then prepared and executed. If an error is encountered, a message string is built with the SQL error code, and the string is passed to the `err_msg` function to be displayed at the terminal. If there have been no errors, the function merely returns.

Figure 9-4 Template Application Delete Error Message

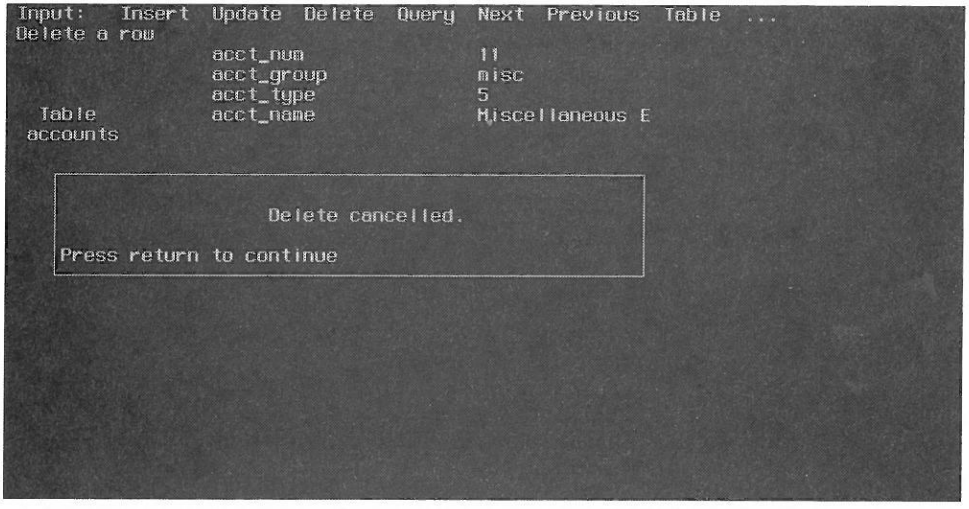
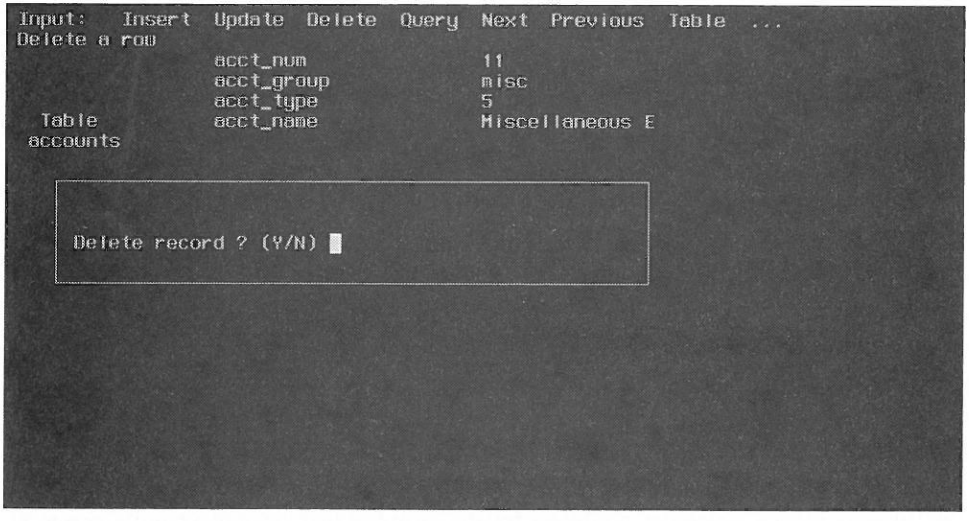


Figure 9-5 Template Application Delete Prompt




```
function gen_delete( tab )define tab char(20)
define qstr char(100)
define msg  char(60)
define retstr char(1)

call prompt_msg( "Delete record ? (Y/N) " )  returning retstr
```

```
if upshift(retstr) != "Y" then
   call err_msg( "Delete cancelled." )
   return
end if

let qstr = " delete from ", tab clipped,
           " where rowid = ", prowid

whenever error continue

prepare _delstmt1 from qstr
if  sqlca.sqlcode < 0 then
    let msg = sqlca.sqlcode using "Error <<< on delete prepare."
    call err_msg( msg )
end if

execute _delstmt1
if  sqlca.sqlcode < 0 then
    let msg = sqlca.sqlcode using "Error <<< on delete execute."
    call err_msg( msg )
end if
whenever error stop

end function
```

The gen_query Function

The gen_query function is used to execute the construct statement on the current database table, but this is not a typical use of the construct statement. The construct statement interacts with a screen form to produce filter criteria for an SQL where clause. In order to create a where clause, the construct statement needs to place column names in the string it will produce. It will get these column names from the names of the fields in the screen form. This is not a problem if the screen form is bound to the database table or tables, but that is not the case with this application. The screen form used in this application is not bound to any database; it is a formonly screen form. The where clause statement produced by the construct statement will appear as follows:

```
" i1 > 100 and i3 < 2000 and i4 = 'Smith' "
```

For all practical purposes, these screen form names will be meaningless when used in a query. In order for a query to work, they must be substituted with the correct field

names for the current table. This is done with the `replace_str` function covered in Chapter 14, which takes the following parameters:

```
replace_str (target_string, replacement_string, search_string)
                returning  query_string, number_replaced
```

This function receives a target string—the string or pattern for which to scan in the search string, the string with which to replace the target string and the string to search or scan. The function returns the resulting query string and the number of strings replaced. The function will only replace a single occurrence of the string and returns the resulting string and the position where the string was found. This function is called as follows to replace the generic field name used in the `gen_tab.per` screen form with the actual name of the table column.

```
call replace_str( "i1", "first_name", "i1 = 'Smith' and ⇐
    i2 = 'Jones'")
                  returning ...
```

This will return a string with the following:

```
" first_name = 'Smith" and I2 = 'Jones' "
```

which is precisely what is needed to create the SQL query.

The `gen_query` function begins by first initializing the arrays that track screen input/output. The `construct` statement is then executed on the generic screen form fields. After the user exits the `construct` statement, the column list is built for the query statement. Since **all** fields on the screen form need to be displayed, every field in the database tables is appended to the screen form list. The screen form names are stored in the `mstr_flds_arr` and are appended to the screen form list in a loop for the total number of columns in the table as stored in the `scr_control` record. The result is a complete SQL `select` list of all the table fields displayed on the screen.

The next `for` loop is used to scan the query string produced by the `construct` statement. As discussed above, this string contains an SQL `where` clause with field names from the generic screen form. These field names must be replaced with the relevant column names. This is done using a `for` loop that will execute an iteration for each column name being displayed on the **current** screen (since those are the only fields for which the user could have entered query criteria). For each of these fields, the `replace_str` function is called to replace the screen form field name with the correct relevant table column name from the `mstr_flds_arr` array. The `replace_str` function is called once for each pattern since the screen form field can only appear in the query string once.

Once the `where` clause part of the query string has been built completely, it is concatenated onto the remainder of the query statement to produce the complete `select` statement. This statement is then returned to the calling function.

```
function gen_query( tab, num_flds)
define tab char(20)
define num_flds smallint
define qstr, fld_list char(500)
define n,x,y,z,len,op_count  smallint
define tstr char(80), op_char char(2), lval char(20)
define fld_count, need_op smallint

message "Enter Query Criteria, then Press <ESCAPE> "
    attribute ( reverse )

--
-- initialize the array and set fields to null
--
call init_flds_arr( MAXFIELDS )
call init_mstr_flds( TOTALFIELDS )

construct by name qstr

on    i1,
      i2,
      i3,
      i4,
      i5,
      i6,
      i7,
      i8,
      i9,
      i10,
      i11,
      i12,
      i13,
      i14,
      i15,
      i16,
      i17,
      i18,
      i19,
      i20
```

```
-- build the field list
for n = 1 to scr_control.total_num_flds -- select *all* of ⇐
     the fields

    if n > 1 then
       let fld_list = fld_list clipped, ","
    end if
    let fld_list = fld_list clipped, mstr_flds_arr[n].⇐
     field_name clipped

end for


let x = 1
while x

  -- build the query portion
  for n = scr_control.curr_scr_start + 1 to ⇐
     scr_control.curr_scr_end

    -- replace form names with field names
    let tstr = ( n - scr_control.curr_scr_start ) using "i<<<"
    call replace_str(tstr clipped,
                     mstr_flds_arr[n].field_name,
                     qstr )
         returning qstr, x

  end for

end while


let qstr = " select rowid,", fld_list clipped, " from ", tab ⇐
     clipped,
          " where " , qstr clipped

return qstr

end function
```

The validate_field Function

The validate_field function is used to validate the data being input. It is called from the gen_input function after each field and, if it returns FALSE, forces the user to remain in that field until valid data is input. It evaluates the data type of the field based on the coltype element of the flds_arr which was loaded from the syscolumns.coltype table. If the column is determined to be a numeric column (coltype is 1, 2 or 3), the contents of the string are evaluated as a numeric character, and a value of TRUE or FALSE is returned to the calling function.

```
function validate_field( fld_num )
define fld_num smallint
define x integer

let x = flds_arr[fld_num].coltype

if flds_arr[fld_num].coltype = 0 then -- --character field
   return TRUE
end if
if flds_arr[fld_num].coltype = 1 or
   flds_arr[fld_num].coltype = 2 or
   flds_arr[fld_num].coltype = 3   then

    return ( is_number( flds_arr[fld_num].val ) )

end if

--
-- Add checking for other data types here
-- could simply execute 'whenever any error', attempt to ⇐
    convert the data type
-- and return the error, if any
--

return ( TRUE )

end function
```

The gen_open Function

The gen_open function takes the query string created in the gen_query function. In this function, the query statement is first prepared. After the statement is prepared, the sqlca.sqlcode is checked for errors. If an error has occurred, an error

message is displayed on the user's terminal and the function returns. If there are no errors during the execution of the `prepare` statement, a `scroll` cursor is declared for the prepared statement and then opened.

The function does not check for errors after the `declare` statement but waits until after the cursor `open` statement. When the function has completed opening the cursor, it returns to the calling function.

```
function gen_open( qstr )
define qstr char(500)
define msg  char(60)

whenever error continue
prepare _qstmt1 from qstr
if sqlca.sqlcode < 0 then
   let msg = sqlca.sqlcode using "Error <<< on query prepare."
   call err_msg( msg )
   return
end if
declare _c2 scroll cursor for _qstmt1
open    _c2
if sqlca.sqlcode < 0 then
   let msg = sqlca.sqlcode using "Error <<< on cursor open."
   call err_msg( msg )
   return
end if
whenever error stop


end function
```

The `gen_fetch` Function

The `gen_fetch` function retrieves the data from the database and loads it into the `mstr_flds_arr` array elements. This function operates on the assumption that the cursor is currently open. If that were not the case, the `fetch` statement would fail and an error would be displayed on the user's terminal screen.

Note that the function attempts to retrieve data into **all** of the elements of the `mstr_flds_arr` array. It is a distinct possibility that the number of columns retrieved will be significantly less than the 60 elements of the `mstr_flds_arr` array; this is not an *error* with INFORMIX-4GL, though it does generate a *warning*. Data will still be loaded into the `mstr_flds_arr` array up to the number of columns in the table. The number of columns in the current table is returned from the `load_fields` function and

is stored in the `scr_control` record; this number is constantly evaluated throughout the application, thus limiting the application to reviewing only the `mstr_flds_arr` array elements that have been loaded with data.

If an error is encountered during the `fetch` operation, an error message is displayed on the terminal screen and the function returns. If a NOTFOUND (sqlca.sqlcode = 100) condition is encountered, an error message indicating that there are no more rows is returned. If the user is attempting to retrieve the first row from a query, an error message indicating that no rows were found is displayed. If no errors are encountered, the function loads the first page of screen data into the `flds_arr` array and returns.

```
function gen_fetch( pos )
define pos smallint
define msg char(60)
define n smallint

whenever error continue
-- fetch all columns
fetch relative pos _c2 into  prowid,
                             mstr_flds_arr[1].val,
                             mstr_flds_arr[2].val,
                             mstr_flds_arr[3].val,
                             mstr_flds_arr[4].val,
                             mstr_flds_arr[5].val,
                             mstr_flds_arr[6].val,
                             mstr_flds_arr[7].val,
                             mstr_flds_arr[8].val,
                             mstr_flds_arr[9].val,
                             mstr_flds_arr[10].val,
                             mstr_flds_arr[11].val,
                             mstr_flds_arr[12].val,
                             mstr_flds_arr[13].val,
                             mstr_flds_arr[14].val,
                             mstr_flds_arr[15].val,
                             mstr_flds_arr[16].val,
                             mstr_flds_arr[17].val,
                             mstr_flds_arr[18].val,
                             mstr_flds_arr[19].val,
                             mstr_flds_arr[20].val,
                             mstr_flds_arr[21].val,
                             mstr_flds_arr[22].val,
                             mstr_flds_arr[23].val,
                             mstr_flds_arr[24].val,
```

```
                                mstr_flds_arr[25].val,
                                mstr_flds_arr[26].val,
                                mstr_flds_arr[27].val,
                                mstr_flds_arr[28].val,
                                mstr_flds_arr[29].val,
                                mstr_flds_arr[30].val,
                                mstr_flds_arr[31].val,
                                mstr_flds_arr[32].val,
                                mstr_flds_arr[33].val,
                                mstr_flds_arr[34].val,
                                mstr_flds_arr[35].val,
                                mstr_flds_arr[36].val,
                                mstr_flds_arr[37].val,
                                mstr_flds_arr[38].val,
                                mstr_flds_arr[39].val,
                                mstr_flds_arr[40].val,
                                mstr_flds_arr[41].val,
                                mstr_flds_arr[42].val,
                                mstr_flds_arr[43].val,
                                mstr_flds_arr[44].val,
                                mstr_flds_arr[45].val,
                                mstr_flds_arr[46].val,
                                mstr_flds_arr[47].val,
                                mstr_flds_arr[48].val,
                                mstr_flds_arr[49].val,
                                mstr_flds_arr[50].val,
                                mstr_flds_arr[51].val,
                                mstr_flds_arr[52].val,
                                mstr_flds_arr[53].val,
                                mstr_flds_arr[54].val,
                                mstr_flds_arr[55].val,
                                mstr_flds_arr[56].val,
                                mstr_flds_arr[57].val,
                                mstr_flds_arr[58].val,
                                mstr_flds_arr[59].val,
                                mstr_flds_arr[60].val

if sqlca.sqlcode < 0 then
   let msg = sqlca.sqlcode using "Error <<< on cursor fetch."
   call err_msg( msg )
   return
end if
```

```
if sqlca.sqlcode = NOTFOUND then
   display "No more rows."
         at 23, 5
         attribute (reverse)
else
   display "" at 23,1
end if

whenever error stop

-- load the 1st page of screen data into the flds_arr array
for n = 1 to MAXFIELDS
   let flds_arr[n].val = mstr_flds_arr[n].val
end for


end function
```

The `gen_disp` Function

The `gen_disp` function takes the data stored in the `flds_arr` array and displays the elements to the screen. For convenience, this function takes all of the elements of the `flds_arr` array and displays them on the screen, even though not all of the elements may have been loaded with data. This can be done since any elements of the array that have not been used have been initialized to NULLs, which will display as a blank on the terminal screen. Upon completion of the display statement, the function returns to the calling function.

```
function gen_disp( tab, num_flds )
define tab char(20)
define num_flds smallint

display flds_arr[1].val,
        flds_arr[2].val,
        flds_arr[3].val,
        flds_arr[4].val,
        flds_arr[5].val,
        flds_arr[6].val,
        flds_arr[7].val,
        flds_arr[8].val,
        flds_arr[9].val,
        flds_arr[10].val,
        flds_arr[11].val,
```

```
        flds_arr[12].val,
        flds_arr[13].val,
        flds_arr[14].val,
        flds_arr[15].val,
        flds_arr[16].val,
        flds_arr[17].val,
        flds_arr[18].val,
        flds_arr[19].val,
        flds_arr[20].val
    to  i1,
        i2,
        i3,
        i4,
        i5,
        i6,
        i7,
        i8,
        i9,
        i10,
        i11,
        i12,
        i13,
        i14,
        i15,
        i16,
        i17,
        i18,
        i19,
        i20

end function
```

The `init_flds_arr` and `init_mstr_flds` Functions

The `init_flds_arr` and `init_mstr_flds` functions initialize the `mstr_flds_arr` and `flds_arr` arrays `val` record element to NULL. They are passed the number of elements to initialize and then use a `for` loop to perform this initialization up to the number of elements passed to the function as a parameter.

```
function init_flds_arr( pflds )
define pflds, n smallint
```

```
for n = 1 to pflds
   let flds_arr[n].val = NULL
end for

end function

-----------------------------------------------

function init_mstr_flds( pflds )
define pflds, n smallint

for n = 1 to pflds
   let mstr_flds_arr[n].val = NULL
end for

end function
```

The ld_flds_val Function

The ld_flds_val function performs the difficult task of keeping the flds_arr and mstr_flds_arr arrays synchronized. In this application, data entry and display are performed using the flds_arr array, which only has 20 elements, the size of the display screen. The data is loaded into the database using the values stored in the mstr_flds_arr array, which has 60 elements, the maximum number of values that can be input.

The function receives parameters for the starting element, the number of elements to process and the direction in which the user is moving. The starting element is determined using the calculation shown below.

```
if ( start_element - num_elements + 1 ) > 0 then
     let i = start_element - num_elements
else
    let i = start_element
end if
```

The ld_flds_val function must move back the number of elements passed into the function as a parameter. This calculation determines whether or not the function can move back that number. For instance, if the function were currently displaying the first screen, its starting element would be one, and the number of elements would be 20. In this case, the function would not be able to move back 20 elements (1 - 20 + 1 = -18 and is NOT > 0) and would instead simply make the starting element one. (This code essentially keeps the application from making an invalid array element reference in code

executed later in the function.) If it is possible to move back the correct number of elements, the application will load an integer with the correct starting element by assigning the integer a value equal to the starting element less the number of elements to process.

Now that a correct start element has been computed, the function must store the current set of values on the screen in the `flds_arr` array in the `mstr_flds_arr` array. This is accomplished using a `for` loop, as shown below.

```
let x = 1
for n = i to ( i + num_elements - 1 )

   let mstr_flds_arr[n].val = flds_arr[x].val
   let x = x + 1

end for
```

This loop will start at the starting element assigned using the expression shown above, and process the 'number of elements to process' value passed into the function. For each iteration of the `for` loop, it will assign a value currently on the screen to the `mstr_flds_arr` array value element. The `mstr_flds_arr` array is the program variable used to update the database. It must be kept current as the user moves between screens since, at any point in time, the user could press the ACCEPT key and update the database.

Next, the direction flag parameter passed into the function is evaluated; if it is set to zero, it is an indication that the function should only be used to load the current elements from the `flds_arr` array into the `mstr_flds_arr` array. If this is the case, the function will return at this point.

If the user is in the process of moving forward a screen page or back a screen page, the `flds_arr` array must be loaded with the values stored in the `mstr_flds_arr` array for the *next* page. This will effectively refresh the `flds_arr` array with the elements that should be displayed on that screen.

If the direction flag passed indicates the function is moving **forward**, the `for` loop starts at the start element parameter passed and continues for the value in the number of elements parameter passed into the function. For each iteration of the loop, a value from the `mstr_flds_arr` array element is loaded into the appropriate `flds_arr` array element.

If the direction flag indicates that the user is moving **back** a page, the starting index for the `for` loop is the result of the expression of starting element parameter less the number of fields to process. The `for` loop will then execute the loop for this starting index to the maximum number of fields that can be displayed on a page (MAXFIELDS).

Once again, for each iteration of the loop, a value from the `mstr_flds_arr` array element is loaded into the appropriate `flds_arr` array element.

```
function ld_flds_val(start_element, num_elements, direction_flag)
define start_element, num_elements, direction_flag smallint
define n,x,i smallint

-- move the current screen flds_arr[].val to mstr_flds_arr[].val
if ( start_element - num_elements + 1 ) > 0 then
     let i = start_element - num_elements
else
    let i = start_element
end if

if ( debug ) then
    message "saving - start ",i, " num_elements ", num_elements
     attribute( reverse)
    sleep 2
end if

-- load current values into the mstr_flds_arr[]
let x = 1
for n = i to ( i + num_elements - 1 )

   let mstr_flds_arr[n].val = flds_arr[x].val
   let x = x + 1

end for

if direction_flag = 0 then
   return
end if

if direction_flag = +1   then
  -- load the next screen's values into the flds_arr
  let x = 1
  for n = start_element to ( start_element + num_elements - 1)
         let flds_arr[x].val = mstr_flds_arr[n].val
         let x = x + 1
  end for
end if

if direction_flag = -1  then
```

```
   -- load the previous screens values into the flds_arr
   let x = 1
   let i = (start_element - num_elements - MAXFIELDS)

   for n = i to ( i + MAXFIELDS - 1)
         let flds_arr[x].val = mstr_flds_arr[n].val
         let x = x + 1
   end for

end if

end function
```

The `ld_flds_lbls` Function

The `ld_flds_lbls` function is used to load the field labels for the screen to be displayed. These field labels are stored in the `mstr_flds_arr` array record and are loaded into the `flds_arr` array record for each applicable element.

The `ld_flds_lbls` function is passed the start element and the number of elements to process. It then performs a loop from the starting element for the number of elements parameter passed. For each loop iteration, a field label is copied from the `mstr_flds_arr` array to the correct `flds_arr` array element.

```
function ld_flds_lbls( start_element, num_elements )
define start_element, num_elements smallint
define n,x,i smallint
-- load field labels for the current screen into the fields ⇐
    array used
-- by the input and display statements

let x = 1
for n = start_element to ( num_elements + start_element - 1 )

     let flds_arr[x].field_name = mstr_flds_arr[n].field_name
     let x = x + 1

end for

end function
```

The move_flds_page Function

The move_flds_page function is used to control screen page movements. This function assures that the appropriate parameters are set correctly and then calls the ld_flds_val and ld_flds_lbls functions.

The move_flds_page function accepts parameters for the name of the table and the direction of the move: +1 for a forward move; -1 for a backwards move. The function then executes one of two if/then statement blocks based on the direction parameter that has been passed.

MOVING FORWARD A PAGE

If the user is moving forward, the first if/then statement block in this function will be executed. Within this control block, there are a number of tests to determine whether or not there is another page to display. First, if the current end of the input screen being displayed is equal to the total number of fields in the table, there are no more screens to display. An error message is displayed that indicates this and the function returns to the calling function.

Next, the current field offset is evaluated against the total number of fields loaded in the program arrays. If the current field offset into the mstr_flds_arr is beyond the total number of fields, an error message is displayed indicating there are no more pages to display.

If the current screen end is less than the total number of fields in the screen, there is another screen to display. The ld_flds_val function is then called with parameter values stored in the scr_control record. The scr_control.curr_fld_offset is the starting position for the current screen. The scr_control.disp_flds record element represents the number of fields displayed on the current screen. These two parameters are passed into the ld_flds_val function. The ld_flds_val function then stores the current screen values into the mstr_flds_arr array and stores the field values for the next screen into the flds_arr array and returns.

The ld_flds_lbl function is then called to load the field labels for the new page, and then the display_form function is called to display the new screen form labels to the screen. Finally, the gen_disp function is used to display the field values.

Before returning, the scr_control record parameters must be reset to reflect the current condition of the screen. Since the program has been asked to move forward one page, the current field offset (stored in the scr_control.curr_fld_offset record element) represents the start of the next screen; this value is therefore assigned to the screen control record element that keeps track of the current screen start element, scr_control.curr_scr_start.

If the current screen start plus the maximum number of fields is greater than the total number of fields, this is an indication that the current page has less than the

maximum number of fields per page. In this case, the current screen end is the total number of fields loaded.

For example, if the current screen start were element 20, the total number of fields 25, and the maximum fields or fields per page 20, the screen start plus fields per page or maximum number of fields would be 40 (20 + 20), which is greater than 25, the total number of fields in the array. The screen end would then be the total number of fields in the current table, or 25.

The number of display fields in the current screen is then computed as the difference between the current screen end and the current screen start. The final calculation sets the current field offset, which is meant to point to the end of the current set of fields. (This number will actually point to the start of the next screen and, in the case where the last screen page is being displayed, this number is allowed to point to an invalid element; error-checking code throughout the application will prevent display errors.)

Moving Back a Page

The application code for moving back a page is similar to the code executed when moving forward a page. First, some code performs validations to make sure that the move will not go beyond limits. When moving back a page, the program must obviously make sure that it has not moved below element zero. Since it is known that if the program is on the first page the current screen start will be set to one, a validation that tests for this and displays an error message if it is true is an adequate validation.

Next, the `ld_flds_val` function is called to load the current screen values into the `mstr_flds_arr` and then load the value for the new page. It is passed the current settings in the screen control record for the current field offset (`scr_control.curr_fld_offset`) and the number of fields displayed on the current screen (`scr_control.disp_flds`).

The next section of code will perform the calculations necessary to move back one screen page. The current field offset less the number of display fields is the start of the current screen. This number less the maximum number of fields or fields per page should be a number greater than zero; if it is not, the current screen start will be set to one (indicating the program is on page one). If the number is greater than zero, this number is assigned as the current screen start.

Next, some other elements of the screen control record are set. The current field offset, which should point to the start of the next forward page, is set by adding the maximum number of fields or fields per row to the current screen start. As long as this number is not beyond the total number of fields in the current table, it will be assigned as the current field offset; if the number is greater than the total number of fields in the table, the total number of fields will be used as the current field offset.

The current screen end is then assigned as the current field offset less one. (This could also have been computed as in the code for moving forward a page, by adding the maximum number of fields to the current screen start.) The number of display fields is assigned as the current screen end less the current screen start.

Finally, the screen field labels for the new screen are loaded using the `ld_fld_lbls` function. This function is passed the current screen start and the number of display fields in the current screen. It will load the `flds_arr` array with the screen field labels from the `mstr_flds_arr` array and then return. The `display_form` function will display the field labels on the screen; it is passed the name of the table and the number of display fields to display. Finally, the `gen_disp` function is called to display the current values in the `flds_arr` array to the screen form.

```
function move_flds_page( tab, move )
define tab  char(20)
define move smallint

if ( move > 0 ) then

    if  scr_control.curr_scr_end = scr_control.total_num_flds⇐
      then      -- no pages to display
      call err_msg( "You are already on the last screen." )
      return
    end if

    -- load the current field values
    --
    -- load the current fields
    call ld_flds_val(scr_control.curr_fld_offset, scr_control.⇐
      disp_flds, +1 )

    -- curr_fld_offset represents the end of the page, not ⇐
      necessarily the end of data fields
    if scr_control.curr_fld_offset >= scr_control.⇐
      total_num_flds then    -- no more pages
      call err_msg( "No more pages to display" )
      let scr_control.curr_fld_offset = ⇐
      scr_control.total_num_flds
      return
    end if

    call ld_flds_lbls( scr_control.curr_fld_offset, ⇐
      scr_control.disp_flds )
    call display_form( tab, scr_control.disp_flds )
```

```
    call gen_disp( tab, scr_control.disp_flds )


    --must now set the scr_control record elements
    let scr_control.curr_scr_start = scr_control.⇐
      curr_fld_offset - 1
    if scr_control.curr_scr_start + MAXFIELDS >⇐
       scr_control.total_num_flds  then
      let scr_control.curr_scr_end = scr_control.total_num_flds
    else
      let scr_control.curr_scr_end = scr_control.⇐
        curr_scr_start + MAXFIELDS
    end if

    let scr_control.disp_flds = scr_control.curr_scr_end - ⇐
      scr_control.curr_scr_start
    -- point this to the end of the list
    let scr_control.curr_fld_offset = scr_control.⇐
      curr_fld_offset + scr_control.disp_flds

end if

if ( move < 0 ) then

    if scr_control.curr_scr_start = 1 then
       call err_msg( "Last Page of Fields" )
       return
    end if

    call ld_flds_val( scr_control.curr_fld_offset,
                  scr_control.disp_flds,    -- load them all
                  -1 )

    -- try to go back one full page

    if ( scr_control.curr_fld_offset-scr_control.disp_flds -⇐
        MAXFIELDS - 1) <= 0 then
      let scr_control.curr_scr_start = 1
    else
        let scr_control.curr_scr_start =⇐
           scr_control.curr_fld_offset - MAXFIELDS -
                                  cr_control.disp_flds
    end if
```

```
    -- load the scr_control record
    -- point the offset at the end of the current page, if it ⇐
        is    not beyond the end of the fields
    if scr_control.curr_scr_start + MAXFIELDS <=⇐
        scr_control.total_num_flds then
        let scr_control.curr_fld_offset =⇐
          scr_control.curr_scr_start + MAXFIELDS
    else
        let scr_control.curr_fld_offset = ⇐
          scr_control.total_num_flds
    end if

    let scr_control.curr_scr_end = scr_control.curr_fld_offset-1
    let scr_control.disp_flds    = scr_control.curr_scr_end -⇐
      scr_control.curr_scr_start + 1
    call ld_flds_lbls(scr_control.curr_scr_start, ⇐
      scr_control.disp_flds )
    call display_form( tab, scr_control.disp_flds )
    call gen_disp( tab, scr_control.disp_flds )

end if

end function
```

The `scr_page` Function

The `scr_page` function is currently designed only to move the application back to the first screen page. The `gen_update` and `gen_delete` functions expect the program to display the first page, and therefore require this function to be called for page one before beginning input. The function is passed the page number and the name of the table. The table name is not evaluated; it is simply passed to additional functions.

First, the current screen start is calculated. Since the page number is one, the current screen start is set to element one. Next, several elements of the screen control record are set. If the total number of fields is less than the maximum number of field per page, this is the value assigned to the number of display fields element in the screen control record. If the total number of display fields is greater than the maximum number of fields, the maximum number of fields is assigned to the display fields element, since these are all the fields that can be displayed on the screen at once. The current field offset is set to the number of display fields plus one, and the current screen end is assigned the value of the current field offset less one.

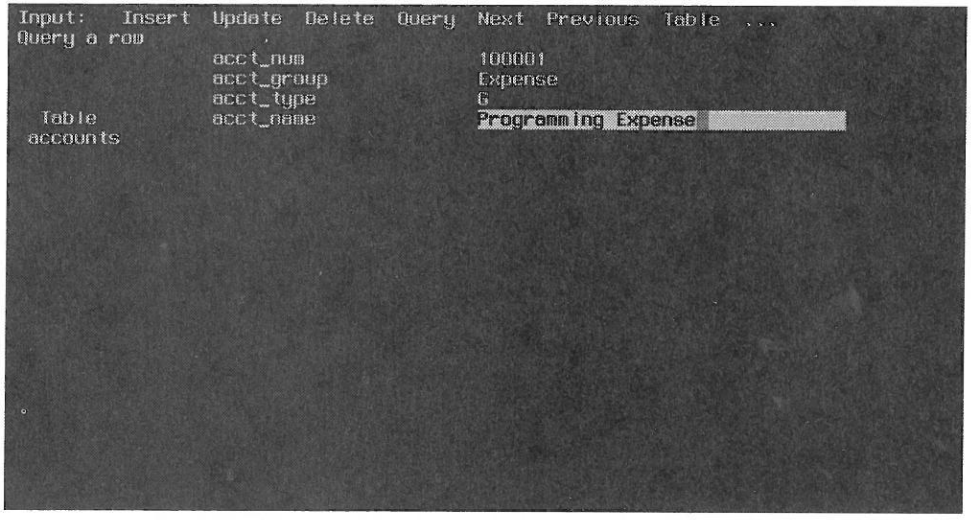
Finally, the ld_flds_lbls function is called to load field labels for the new page. The display_form function is called to display the field labels to the screen, and the gen_disp function is called to display the values currently contained in those fields.

```
function scr_page( tab, page_num )
define tab char(20)
define page_num smallint

if page_num = 1 then
   let scr_control.curr_scr_start = 1
end if

if scr_control.total_num_flds < MAXFIELDS  then
   let scr_control.disp_flds = scr_control.total_num_flds
else                              -- we have multiple pages
   let scr_control.disp_flds = MAXFIELDS
end if

let scr_control.curr_fld_offset = scr_control.disp_flds + 1
let scr_control.curr_scr_end    = scr_control.curr_fld_offset - 1

call ld_flds_lbls( scr_control.curr_scr_start,⇐
     scr_control.disp_flds )
call display_form( tab, scr_control.disp_flds )
call gen_disp( tab, scr_control.disp_flds )


end function
```

The gen_input Function

The gen_input function performs the input of data from the screen (see Figure 9-6) to the program variables. It performs a series of before field/after field statements which provide highlighting of the current field, data validation and enforcing the end of the input list.

Figure 9-6 Table Browser Input Screen



The function contains an `options` statement, which selects the `input wrap` mode of input. This will require the user to explicitly make use of the accept key to end input. The `gen_input` function is called by the `gen_add` or `gen_update` function. If it has been called by the `gen_add` function, it will be passed a parameter which will be set to the string "ADD". If the function is in ADD mode, the `flds_arr` and `mstr_flds_arr` arrays are cleared of their current values using the `init_flds_arr` and `int_mstr_flds` functions, respectively.

Next, the `num_flds` parameter is incremented to coincide with the logic of the field validations. Then the `input` statement is started. First within the input block are the function key/control key traps. The Control-F key is used to allow the user to move forward a page and the Control-B key allows the user to move back a page. These `on key` blocks trap the keys and call the `move_flds_page` function to move the page forward or back one page. After moving the screen a page, they use the `next field` statement to position the cursor in the first field on the screen form.

This section of code has some repetition due to the nature of the before field/after field clause. The current implementation of `before field/after field` blocks require a string character **constant** as a parameter. This leads to the requirement that each `before field/after field` block be a separate section of code.

In each of the `before field` blocks, the application code performs a calculation to determine whether or not the user is attempting to move beyond the current end of the input screen. If the user is not pressing the UP key or the ACCEPT key, implying that they are attempting to move down in the screen field list, they are quietly moved to the first field in the screen form; this has the effect of emulating the `input wrap` option.

Also within the `before field` block, the current field is displayed in the reverse attribute.

In the `after field` block, if the `gen_input` function has been called with the validate flag set to TRUE, the validate routine is called with the current field number. If the `validate_field` routine returns false, an error message is displayed and the cursor remains in the current field. Before the control block ends, the current field is displayed in normal video, thus turning off the reverse video.

After the end of the input block, the data that is currently loaded into the `flds_arr` array is loaded into the `mstr_flds_arr` array. A `for` loop that moves from the current screen start element to the current screen end element is used to accomplish this. The `gen_input` function then returns to the calling function.

```
function gen_input( tab, num_flds, validate_flag, imode )
define tab char(20)
define num_flds smallint
define validate_flag smallint
define n,x smallint
define imode char(10)

options input wrap

if imode = "ADD"  then
   call init_flds_arr( MAXFIELDS )
   call init_mstr_flds( TOTALFIELDS )
end if

let num_flds = num_flds + 1

input  flds_arr[1].val,
       flds_arr[2].val,
       flds_arr[3].val,
       flds_arr[4].val,
       flds_arr[5].val,
       flds_arr[6].val,
       flds_arr[7].val,
       flds_arr[8].val,
       flds_arr[9].val,
       flds_arr[10].val,
       flds_arr[11].val,
       flds_arr[12].val,
       flds_arr[13].val,
       flds_arr[14].val,
       flds_arr[15].val,
```

```
        flds_arr[16].val,
        flds_arr[17].val,
        flds_arr[18].val,
        flds_arr[19].val,
        flds_arr[20].val
 without defaults
 from   i1,
        i2,
        i3,
        i4,
        i5,
        i6,
        i7,
        i8,
        i9,
        i10,
        i11,
        i12,
        i13,
        i14,
        i15,
        i16,
        i17,
        i18,
        i19,
        i20


 -- ON KEY clauses

    on key (Control-F)

       call move_flds_page( tab, +1 )
       next field i1

    on key (Control-B)

       call move_flds_page( tab, -1 )
       next field i1


 --
 -- emulate input wrap
```

```
--

   before field i1
     if scr_control.disp_flds + 1  = 1  and
        fgl_lastkey() <> fgl_keyval("ACCEPT") and
        fgl_lastkey() <> fgl_keyval("UP")        then
        next field i1
     end if
     display flds_arr[1].val to i1 attribute ( reverse )

   after field i1
     if validate_flag  then
       if validate_field( 1 )  then
          display flds_arr[1].val to i1
       else
          error "Invalid input."
          next field i1
       end if
     end if
     display flds_arr[1].val to i1

   before field i2
       if  scr_control.disp_flds + 1  = 2 and
           fgl_lastkey() <> fgl_keyval("ACCEPT") and
           fgl_lastkey() <> fgl_keyval("UP")        then
         next field i1
       end if
       display flds_arr[2].val to i2 attribute (reverse)

  after field i2
     if validate_flag then
       if validate_field( 2 ) then
          display flds_arr[2].val to i2
       else
          error "Invalid input."
          next field i2
       end if
     end if
     display flds_arr[2].val to i2

..... code deleted ... it repeats up to 20 .....

end input
```

```
-- load the current data in flds_arr into mstr_flds_arr
let x = 1
for n = scr_control.curr_scr_start + 1 to
        scr_control.curr_scr_end

    let mstr_flds_arr[n].val = flds_arr[x].val
    let x = x + 1

end for


end function
```

SUMMARY

The table browser application demonstrates the ability of INFORMIX-4GL to be used for dynamic, generalized applications. This application makes use of dynamic SQL queries, screen forms that are not bound to the database, and multiple screen forms. All are packaged together in an application that can be used with virtually any database table. It is code that can be used as a whole application, or can be embedded in an existing application (the application is initiated with a single function call and uses only a few global variables).

There are limitations with the current version of this application. Data type validation is weak and large screen fields are not supported. Blob data types are also not supported. Given the modularity of the application, data validation could be added by adding validation code to the `gen_validate` function. Supporting large screen fields would be more problematic, since the application currently works on the assumption that the maximum size of a screen field is 60 bytes. And support for Blobs, as always, is even more difficult.

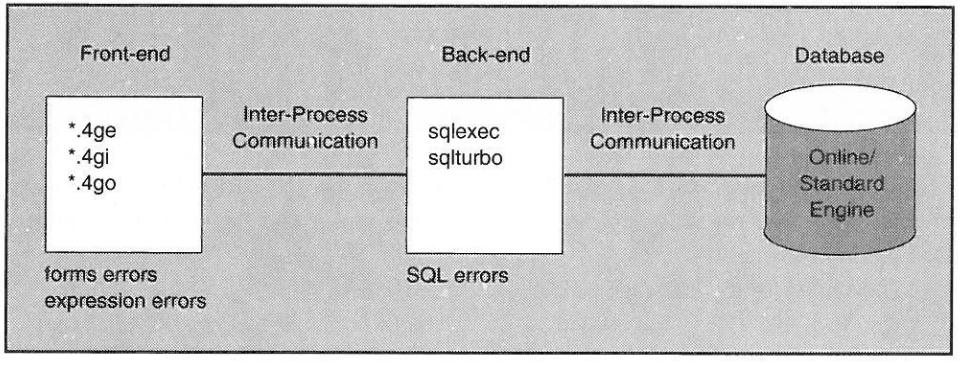
Chapter 10

Error Handling and Help Messages in INFORMIX-4GL

INFORMIX-4GL provides several different methods for error trapping and the display of error messages and help screens. Some are intended to help the programmer during the debugging phase of application development, and others are intended to provide useful, informational error messages for the user at runtime.

INFORMIX-4GL provides three types of error messages: SQL errors, INFORMIX-4GL language syntax errors and forms errors. INFORMIX-4GL language syntax errors are trapped by the compiler. Errors that refer to SQL syntax are also checked at compile time, with the exception of SQL statements *prepared* using the SQL `prepare` statement; these statements are invisible to the compiler. The syntax for these statements is not checked until runtime, and then they are checked by the engine, not by the INFORMIX-4GL front end.

Figure 10-1 Front-End and Back-End Error Processing

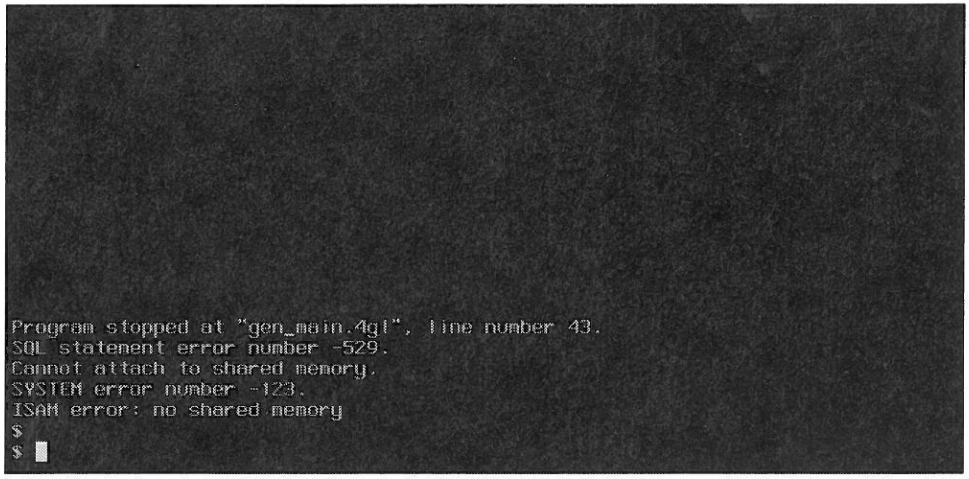


Forms errors include both errors that occur during screen form interaction as well as runtime errors that occur with a variety of expressions. This includes errors such as division by zero, decimal overflow and date/date-time operations. Forms errors are *not* trapped by the simple `whenever error` statement; the `whenever any error` statement must be used.

whenever error STATEMENT

INFORMIX-4GL provides very good default error reporting. This error report includes the SQL error number that occurred, if any, the source code module and line number where the error occurred, and a brief description of the error. By default, the INFORMIX-4GL runtime component will display this error message and then terminate the program (see Figure 10-2). This method of error trapping, which may not be the preferred method to use in a production environment, is very useful during the debugging process.

Figure 10-2 Default INFORMIX-4GL Error Reporting


Program stopped at "gen_main.4gl", line number 43.
SQL statement error number -529.
Cannot attach to shared memory.
SYSTEM error number -123.
ISAM error: no shared memory
$
$

In order to override the default error checking, the INFORMIX-4GL language provides the `whenever error` statement to trap errors and trigger various events based on these errors.

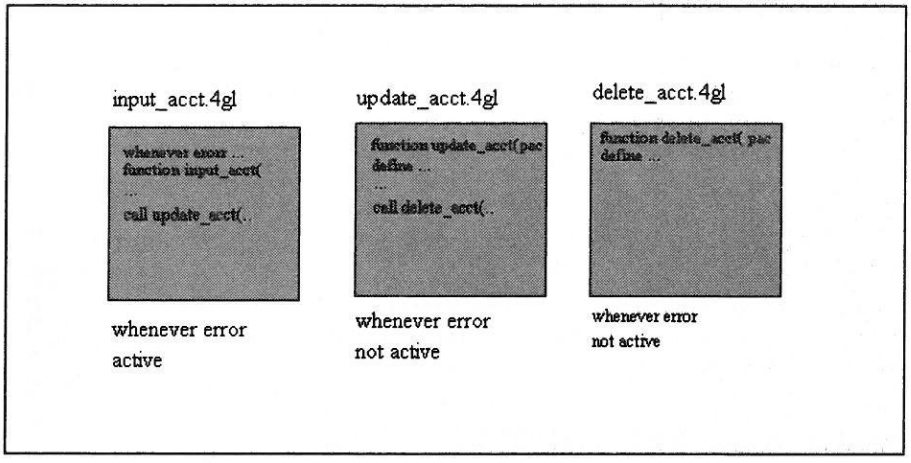
The syntax for this message is:

```
whenever | any | error | stop | continue | goto label | call⇐
     function
```

The `whenever error` statement has module scope. It can be thought of as a *compiler directive* since its impact is controlled by the compiler. Upon encountering this statement, the compiler will begin to generate the appropriate code to execute the method of error handling chosen, serially, from the point where the `whenever error` statement was encountered in the source code module. The impact of the `whenever error` statement is therefore from the point at which it is encountered to the next

occurrence of a `whenever error` statement or the end of the source code file (see Figure 10-3).

Figure 10-3 The Scope of the `whenever error` Statement



The `whenever error continue` method of error trapping will, as the statement implies, continue on receipt of an error. If this method of error handling is used, then following the line on which an error occurred, the global status variable will be set to the error number of the error that occurred. It is the programmer's responsibility to examine this variable and respond accordingly. Generally, because of the need to continually check the status variable after statements, it is advisable to limit the scope of `whenever error continue` statements and provide some other consistent method of error trapping in production applications. The sample code below demonstrates a common use of the `whenever error continue` statement.

```
function get_data( rec1 )
define rec1 like table1.*

whenever error continue
select *
into rec1.*
from table1
where table1.col1 = rec1.col1

whenever error stop

if sqlca.sqlcode < 0  then
    error "Error ", sqlca.sqlcode, " on select "
```

```
    initialize rec1.* to NULL
  end if

  return rec1.*

  end function
```

In this example, a function is called to retrieve data from the database. The `whenever error continue` statement is placed in the code before the SQL statement is executed. Immediately after the SQL statement, default error checking is restarted using the `whenever error stop` statement. When the SQL statement has been executed, the `sqlca.sqlcode` is examined to determine whether or not the SQL statement succeeded. If it did not succeed, an error message is displayed on the terminal screen using the **error** statement, and the return record is set to NULL to indicate to the calling function that no data was retrieved. The final line of the function returns the record to the calling function.

ERROR LOGGING WITH INFORMIX-4GL

Informix-4GL provides a set of functions to allow error messages to be logged to an ASCII file. This method of error logging has the benefit of storing errors that may occur during the running of a production application. Since end users are notorious for either not noticing or misreading error messages displayed on the terminal screen, with error logging error messages can be stored for later retrieval and review by technical staff. Error logging also retains a historical list of errors that have occurred, providing a useful debugging tool.

The error log is started using the `startlog` function. Once started, successive errors that occur during the program will produce entries in this file. Entries will contain the date, time, source module name, line number, error number and error message. Errors are always *appended* to the error log file, making it the programmer's responsibility to occasionally clear this file.

Entries can also be specifically written to the error log file using the **errorlog** function. The following code demonstrates the use of the INFORMIX-4GL error-logging facilities.

```
main

...

call startlog( "prog1.err")

call get_data( key1, key2 )
```

```
    returning rec1.*
...

end main

--------------------------------------------------------------------

function get_data( key1, key2 )
define rec1 like table1.*

-- allow errors to be logged; continue program execution on error
whenever error continue

select col1, col2
into     rec1.*
from   table1
where col1 = key1  and
           col2 = key2

whenever error stop

-- write entry to error log indicating function call that failed
-- display error message to user just telling user the data ⇐
    is not there
if sqlca.sqlcode < 0   then
    errorlog( "The get_data function call failed " )
    error  "Data not available."
                -- display to the terminal screen
    initialize rec1.* to NULL
end if

return rec1.*

end function
```

In this program, the `startlog` function is first called to create the error log in the **current** directory (passing a full path name to the `startlog` function would have created the error log in a specific directory). The function `get_data` is used to retrieve data from the database. When the `get_data` function is called, it first turns off the default error checking and instructs the program to continue when errors are encountered. Since error logging is already turned on at this point, the program will write error information to the error log and continue execution.

In this application, if an error is encountered during the `select` statement, an error will be logged to the error log. After the `select` statement, the `sqlca.sqlcode` is evaluated; if it is less than zero, an error has occurred and additional descriptive information is written to the error log. This will provide additional information to the technical reviewer of the error log.

TRAPPING FORMS AND EXPRESSION ERRORS WITH `whenever any error`

Forms errors, which covers both screen form input/output errors and expression errors, are not trapped by the simple form of the `whenever error` statement; the **any** keyword must be used in the `whenever any error` statement. The code below demonstrates the use of this statement.

```
function fix_data( str1, int1, int2 )
define str1 char(10)                -- should be a date
define int1, int2 integer           -- divide and return the result
define dte date
define result decimal

whenever any error continue

-- create the date

let dte = date( str1 clipped )
if status < 0 then
   error "Error in date conversion. Date string is ",str1
   let dte = NULL
end if

-- perform the division
let result = int1 / int2
if status < 0 then
   error "Error in division. Operands: ",int1, " / ", int2
   let result = NULL
end if

whenever any error stop

return dte, result

end function
```

In this example, the function `fix_data` must convert a string into a date, and divide two integers and return the result. The `whenever any error` trapping provides a convenient method of trapping for errors in expressions. In this case, the `whenever any error` statement uses the `continue` clause to allow program execution to continue on expression errors and let the code after the expression check for errors and respond accordingly.

In the case of the string to date conversion, the expression is evaluated in the form of a function call to the `date` function. After the function call, the `status` variable is checked for a negative value. If the value is negative, a date conversion error is assumed, and an error message is displayed which indicates that an error has occurred in date conversion and displays the offending date string. The `date` variable is then set to NULL.

In the mathematical expression, two integer numbers are divided and the result stored in the result variable. A potential error in this expression is the divide by zero expression error. Expression errors are trapped by examining the status code after the division operation; if it is less than zero, a division error is assumed to have occurred, and an error message is displayed with the offending operands. The **result** variable is then set to NULL.

Note that these examples do not check for specific error codes. It is asserted that if an error occurred, it occurred on the line above where the status code is examined. This is a fairly safe assumption in the above example, but may not always be the case. Placing the `whenever any error` statement close to where the expression error may occur can decrease the likelihood that an incorrect error could be trapped

If a specific error number were to be trapped, a `case` statement with several error numbers would have to be placed in the code. In many cases, it may be difficult to tell specifically which error number or numbers to use. Informix documentation does not always specify which error numbers are returned by failed library function calls or expressions. For these reasons, the error-trapping method shown above is the easiest form to use.

ERROR MESSAGES

Error messages can be displayed using several methods. The most common of these is the `error` statement. The `error` statement takes a single string argument, rings the terminal bell and displays the string on the error line (defined in the `options` statement) in reverse video. The current version of the `error` statement does not accept expressions as arguments, only simple data types.

In addition to the `error` statement, there are several functions related to error handling that can be used to display error messages. They are listed in Table 10-1.

Table 10-1 INFORMIX-4GL Error Handling Functions

Function	Purpose
`err_print`	Print the error message on the error line in reverse video.
`err_get`	Takes the error number as an integer argument and returns a character string error message that corresponds to the error number.
`err_quit`	Takes the error number as an integer argument and prints the error message on the error line and then quits the program.
`errorlog`	Takes a string as an argument and prints the string in the error log along with the date and time the message was printed. This requires that the `startlog` function has been called previously.
`startlog`	Takes a string argument that corresponds to the name and/or full path of the error log file. If a full path is not given, the error log is opened in the current directory. If a full path name is given, the error log is opened in the directory specified in the full path.

GENERALIZED ERROR-HANDLING ROUTINES

It is often desirable to have a generalized error-handling routine available for programmers working in a modular application development environment. When necessary, application code can turn off the error handler using the `whenever error continue` statement, and then turn the error handler back on using the `whenever error` statement. The following code demonstrates this method.

```
function err_handler()

whenever error call err_handler
...

function err_handler()

call  errorlog( err_get )
call err_print( sqlca.sqlcode )

end function

------------------------------------------------------------------

function get_count( key1, key2 )
define key1 like table1.col1
```

```
define key2 like table1.col2
define cnt integer

-- turn off the default error handler
whenever error continue
select count(*)
into   cnt
from   table1
where  col1 = key1 and
       col2 = key2

if sqlca.sqlcode < 0 then
   let cnt = 0          -- rather than null
   call err_handler()   -- now call this routine to display ⇐
    the error message
end if
whenever any error call err_handler

return cnt

end function
```

In this example, the `whenever error` statement is used to trap all errors with the `err_handler` function. All errors are written to the error log file and then an error message is printed to the terminal screen. This is accomplished using the `errorlog` function, which has as its argument the return value of the `err_get` function. Since the `err_get` function has been called with the error number of the current error, it will return an error message corresponding to the current error; this is the error message that will be written to the error log. The final line of the `err_handler` function calls the `err_print` function with the error number of the current error. This will print the error message for the current error to the terminal screen.

An Error Prompt Function

One minor weakness of the error handling facilities is that they do not ensure that the user has read them. The message is merely displayed on the screen and/or written to the error log. There is no validation that the user has read the error message.

A dialog box which requires the user to press a key to continue working ensures that the user has at least seen the error message (and, in most cases, has not written the message down). The following code demonstrates the use of a function that opens a window, displays information on the error that has occurred, and then waits for the user to press return before continuing with the program.

```
function err_msg( msg )
define msg    char(80)
define tmpmsg char(60)
define c1     char(1)

let msg = center( msg, 50 )

open window _ew1 at 10, 5 with 4 rows, 50 columns
          attribute( border, prompt line last )

display msg clipped at 2,2 attribute ( bold )

prompt "Press return to continue " for char c1

close window _ew1

end function
```

This function accepts a single parameter: the message string. It then opens a window, displays the error message using the bold attribute and then prompts the user to press return to continue with the program. After the user presses the return key, the window opened is closed and the function returns. The programmer can still choose to end the program at this point, but this function at least requires some interaction from the user, forcing the user to acknowledge the problem.

HELP MESSAGES IN INFORMIX-4GL

INFORMIX-4GL provides a complete set of help facilities to provide end-user help at runtime. Once a help file has been created using the correct format, it can then be displayed on the screen automatically with certain INFORMIX-4GL statements, or explicitly using the `showhelp` function. Help messages are created from a delimited text file using a utility called `mkmessage`. The contents of this text file are entries as follows

```
.100

This is the error message 100.  When the showhelp
function is called with and integer parameter 100, this
message will be displayed to the screen.

.200
```

```
This is the error message 200.  When the showhelp
function is called with and integer parameter 200, this
message will be displayed to the screen.
```

The help file is created using a text editor and should be composed of only simple ASCII characters. It is compiled using the `mkmessage` utility, which takes arguments as follows.

```
mkmessage input_file output_file
```

This utility does **not** assume or enforce any naming convention for the output file. It will take the input file, process it and produce the output file. Beware: if the output file is given the same name as the input file, the output file will be produced from the input file and the output will *overwrite* the original input file quietly, without complaint.

Help messages can be used with certain statements, such as the menu statement or prompt statement. The default key for triggering on-line help with these statements is Control-W, but this can be reassigned using the `options` statement. The following code demonstrates the use of help in INFORMIX-4GL.

```
options help file "menu.hlp"

menu "Input"

      command "Insert"
              help 101

      command "Other"
              call showhelp( 201 )

...
```

In this code fragment, the `menu` statement is used with the `help` clause. Note that the options statement is used to identify the help file. The help file must be identified before being referenced; if not, a runtime error will occur. The `help` clause takes an integer parameter corresponding to the help number for that menu option. When the user presses the help key while the menu is displayed and the cursor rests on that option, the help message corresponding to help key 101 will be displayed. This is an example of implicit help.

The `menu` command for the 'Other' menu option has an example of an explicit help message. If this option is chosen, then the function `showhelp` is called to display the help message for the integer help number that has been passed.

This form of explicit help message is frequently used to provide field-level help with the `input` statement. The `input` statement provides for a `help` clause, but it is for the statement in general, not for the specific fields in the `input` statement. In order to provide field-level help, the following strategy is often used.

```
options help file "input.hlp",
              help key  Control-T
...
input rec1.* from scr_rec.*

     on key (Control-W, F1 )
               case
                   when infield( col1 )
                           call showhelp( 101 )
                   when infield( col2 )
                           call showhelp( 102 )
                   when infield( col3 )
                           call showhelp( 103 )
        ....
```

In this example, the `options` statement is first used to identify the name of the help file and to redefine the standard help key to another key. Next, the input statement is started. When the control key W is pressed, the `case` statement will be executed. Within the `case` statement, each field is tested using the `infield` function to determine if that is the current field. Depending on the current field, the `showhelp` function is then called to display the help message for that field.

THE `sqlca` RECORD

INFORMIX-4GL tracks errors through the `sqlca` (SQL communication area) record. This record is used to record both errors and warnings. Errors are generally unexpected events that occur during processing and could adversely affect the application's operation. Warnings are unexpected events that could be considered acceptable during program operation. The distinction between an error and a warning is sometimes nebulous. A programmer must be aware of these distinctions and respond accordingly.

Errors can be returned by the front end or back end. Back-end or engine errors occur during SQL processing and are returned by the engine at runtime. When an SQL error occurs, the status variable is set to the value of the SQL error code, and the `sqlca` record is populated with additional information about the error. The elements of the `sqlca` record are as follows. Table 10-2 lists value ranges of SQL error codes.

Example 10-1 The **sqlca** Record Structure

```
define sqlca record
                        sqlcode integer,
                        sqlerrm     char(71),
                        sqlerrp     char(8),
                        sqlerrd     array(6) of integer,
                        sqlwarn     char(8)
end record
```

Table 10-2 SQL Error Code Value Ranges

Error Code	Meaning
1 - 99	Operating system errors.
1- 200	C-isam/rsam errors. These are low-level errors that occur. Usually, `sqlca.sqlerrd[1]` is set for the isam error code, if any. This can supply important information about why an operation failed.
0	If SQL operation was successful, this code will be set by the engine. All other elements of the record are set dependent on the SQL operation that was being executed.

THE sqlerrd ARRAY

The `sqlca.sqlerrd` element of the `sqlca` record is an array of six integers that contain information about the last SQL operation executed by the engine. Elements contain information as shown in Table 10-3.

Table 10-3 `sqlerrd` Array Elements

`sqlerrd` Element	Purpose
1	Not used at this time.
2	Serial value returned if SQL operation was successful and involved an insert into a table with a serial column. If an SQL error occurred, `sqlca.sqlcode` < 0, this is set to the isam error that occurred.
3	The number of rows processed. If an update statement has been executed and the database is not in ANSII mode, this is an indication of whether or not the update succeeded.
4	The estimated CPU cost for the query.
5	The offset of the SQL error into the SQL query statement processed.
6	The `rowid` of the last row selected.

THE `sqlwarn` STRING

The `sqlwarn` element of the `sqlca` record is a character string composed of single characters that indicate the warning generated by the last SQL operation. This string contains useful information on the type of database opened, whether or not logging is being used in the database and whether or not the last SQL statement was an Informix ANSII extension. Elements of this string are set as shown in Table 10-4.

Table 10-4 `sqlwarn` Elements

String Element	Purpose
1	Set to W if one of the other array elements has been set. Interrogating this element effectively provides information on whether or not a warning has occurred.
2	On execution of a database open statement, this is set to W if the database is using transaction logging. If a Select statement has been executed and a character string had to be truncated to fit in an INFORMIX-4GL character variable, this element is set to W. (This does not indicate which variable has been truncated. INFORMIX-4GL currently does not provide a means of doing this.)

continued on next page

Table 10-4 continued

String Element	Purpose
3	On execution of a database open statement, this is set to W if the database opened is an ANSI mode database. If a `select` statement has been executed, and an aggregate function within the `select` statement has encountered a NULL, this element is set to W.
4	On execution of a database open statement, this is set to W if the database opened was an Informix-On-line database. If a `select` statement has been executed, this element is set if the number of items in the select list does not match the number of INFORMIX-4GL variables provided to receive the selected data.
5	This element is set to W if float to decimal conversion is used.
6	If the last statement executed as an Informix ANSI extension **and** the DBANSIWARN environment variable is set, this element is set to W.
7	Not used.
8	Not used.

Front-end errors are expression errors that could not be trapped at compile-time, such as division by 0 or numeric overflow. In order for these to be trapped by the application, the `whenever any error` statement must appear in the application before the code that must trap the error.

SUMMARY

INFORMIX-4GL provides a useful set of tools for the programmer to make applications respond gracefully to errors. It is up to the programmer to implement these facilities since the INFORMIX-4GL default error checking is probably *not* the preferable method of error checking in a production environment.

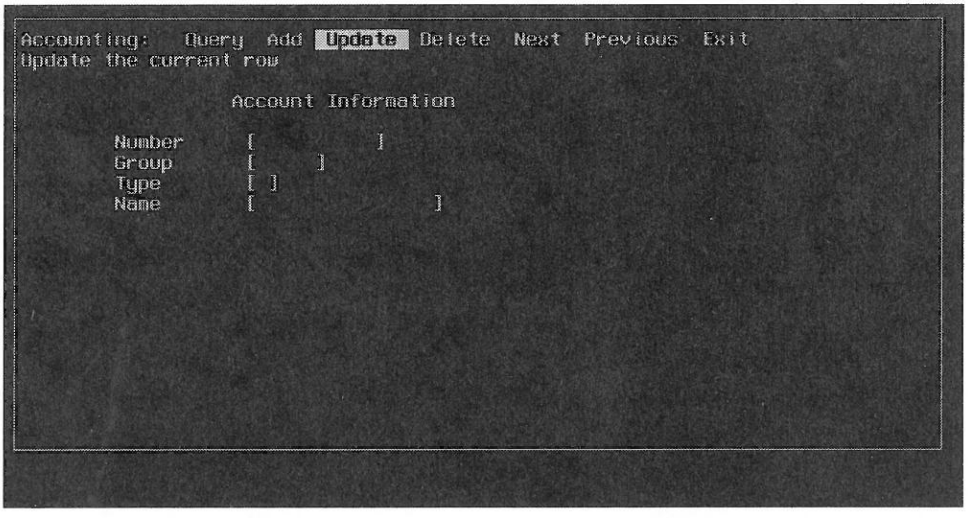
Making use of a centralized, generalized error function can make the application more consistent and reduce the amount of coding necessary to develop the application. Such a modular approach will have both short-term and long-term benefits for the application development staff. In the short term, the application will be developed more quickly. In the long term, the application will be easier to modify because nearly all error checking will be maintained in one place.

Chapter 11

An INFORMIX-4GL Template Application

A template is a working program that has been designed to allow for easy modification. It is designed to be either modified or used as a blueprint for creating a new application. It is, to a certain extent, the standard by which other programs with a similar purpose will follow (see Figure 11-1).

Figure 11-1 Template Application



Use of templates helps to create a modular development environment. Programmers will, by default, use a similar format and flow-of-control structures within their application. They will be able to perform the same amount of work, develop an application, in a shorter amount of time since much of the code will already have been developed for them.

The following example presents a template input and modification application. The application makes use of many of INFORMIX-4GL's features such as the construct statement to perform query-by-example, scroll cursors and the ring menu. It also makes use of a library function presented in Chapter 14 to allow the user to *zoom* into a field and use a pick list to retrieve information from a related table. All of this is accomplished without the use of global variables and with relatively few lines of code.

THE `main` PROGRAM BLOCK

The `main` program block in the template application opens a bordered window which will encompass most of the screen. A screen form which will be used to perform data entry is then opened and the main menu for the application is displayed.

The `appl_name` string is assigned the application name, which is then displayed as the menu title in the `menu` statement. The `whenever error` statement is called to direct program execution to the `err_handler` function whenever an error occurs.

Before the main menu loop begins, the `active_rec` flag is set to FALSE. This flag is used to indicate whether or not there is currently an open cursor being used to browse the table or tables used in the application.

The main menu displays options for query, add, update, delete, next, previous and exit. This menu is the main loop for the application; control will remain in this loop until the user chooses the exit option from the menu.

The QUERY Option

The **query** option allows the user to query the database using query-by-example. First, the `active_rec` flag is set to FALSE to indicate that the query has not yet been successful. The function `qbe` is then called to perform query-by-example using the `construct` statement and return the resulting query string to the `qstr` variable. The `open_curs` function is then passed the `qstr` string containing the query criteria.

The `open_curs` function will use the query string to build the complete query statement. It will then `prepare`, `declare` and `open` the cursor for the query defined in the query string. The `open_curs` function will open the cursor and return the first record from the cursor set. The record is stored in the `appl_rec` record, which is then passed to the `display_rec` function to display the contents of the record to the screen.

If, on return from the `open_curs` function, the `sqlca.sqlcode` is not equal to zero, the menu loop is continued. Since the `active_rec` flag has been set to FALSE, this will disallow many of the other menu options until the `open_curs` function has been called successfully.

```
command "Query" "Query for the current row"
        let active_rec = FALSE
        call qbe() returning qstr
        call open_curs( qstr ) returning appl_rec.*
        if sqlca.sqlcode != 0 then
           continue menu
        end if
        call display_rec( appl_rec.* )
        let active_rec = TRUE
```

The ADD Option

Since the purpose of the ADD option is to create a new record, there is no need to check the setting of the active_rec flag. This option will first call the input_rec function, passing the function a flag to indicate that input should be done in ADD mode. The input_rec function will return the application record, which will then be passed to the add_rec function. The add_rec function will insert the record into the database.

```
command "Add" "Insert a new row"
        call input_rec( appl_rec.*, "ADD" ) returning appl_rec.*
        call add_rec( appl_rec.* )
```

The UPDATE Option

If the active_rec flag is set to TRUE, the command block for the UPDATE option calls input_rec function with a flag indicating that the input will be done in MODIFY mode. The input_rec function then returns the appl_rec record, which is then passed to the update_rec function.

If the active_rec flag is not set to TRUE, an error message is displayed, indicating to the user that they must query for a record first, and the menu loop continues.

```
command "Update" "Update the current row "
          if active_rec then
             call input_rec( appl_rec.*, "MODIFY" )
                    returning appl_rec.*
             call upd_rec( appl_rec.* )
          else
                 call err_msg(
                    "You must Query for a record first." )
          end if
```

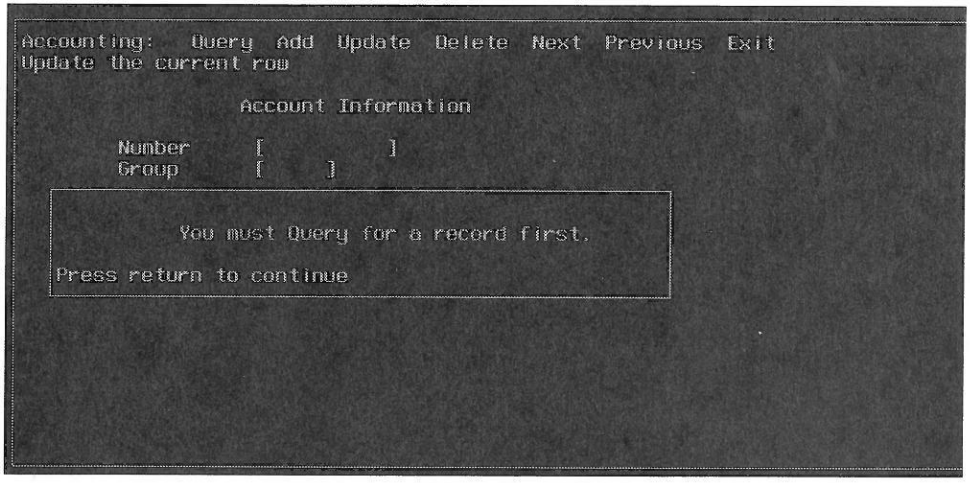
The DELETE Option

If the active_rec flag is set to TRUE, the command block for the DELETE option simply calls the delete_rec function with the current record. The delete_rec function then quietly deletes the record.

If the active_rec flag is not set to TRUE, an error message is displayed, (see Figure 11-2) indicating to the user that they must query for a record first, and the menu loop continues.

```
command "Delete" "Delete the current row"
        if active_rec then
           call delete_rec( appl_rec.* )
        else
           call err_msg( "You must Query for a record first." )
        end if
```

Figure 11-2 An Error Message in the Template Application


Accounting: Query Add Update Delete Next Previous Exit
Update the current row
Account Information
Number [ ]
Group [ ]
You must Query for a record first.
Press return to continue

The NEXT Option

The NEXT option allows the user to move to the next row in the current set. If the `active_rec` flag is set to TRUE, the `curs_move` function is called with a positive integer one. This function call will return the application record into the `retr_rec` record.

If the key field in the `retr_rec` record is NULL, it indicates that the fetch operation failed and the remainder of the record has been set to NULL. In this case, control will fall through to the menu loop and the menu loop will continue.

If the key field in the `retr_rec` is not NULL, the contents of the `retr_rec` record are assigned to the `appl_rec`, and the `display_rec` function is called to display the record. It is necessary to move the data retrieved into the `appl_rec` because all other options in the menu loop expect this record to contain the current data record. The code for this menu option is shown below.

```
command "Next" "Move to the next row"
        if active_rec then
           call curs_move( +1 ) returning retr_rec.*
```

```
            if retr_rec.acct_num IS NOT NULL then
               let appl_rec.* = retr_rec.*
               call display_rec( appl_rec.* )
            end if
         else
            call err_msg( "You must Query for a record first." )
         end if
```

The PREVIOUS Option

The PREVIOUS option contains code identical to the NEXT option, with the exception of the call to the `curs_move` function. In the PREVIOUS option, this function is passed a negative one parameter, indicating that it is moving back one row.

The EXIT Option

The EXIT option merely exits the menu. This will effectively terminate the application.

The complete code for the `main` program block for the template application is shown below.

```
main
define appl_name char(20)
define appl_rec, retr_rec record like accounts.*
define qstr char(120)
define active_rec smallint

let appl_name = "Accounting "

call init_vals()

whenever error call err_handler

open window w1 at 2,2
   with 20 rows, 78 columns
   attribute( border )

call open_form()

let active_rec = FALSE

   menu  appl_name

      command "Query" "Query for the current row"
```

```
      let active_rec = FALSE
      call qbe() returning qstr
      call open_curs( qstr ) returning appl_rec.*
      if sqlca.sqlcode != 0 then
         continue menu
      end if
      call display_rec( appl_rec.* )
      let active_rec = TRUE

   command "Add" "Insert a new row"
      call input_rec( appl_rec.*, "ADD" ) returning appl_rec.*
      call add_rec( appl_rec.* )

   command "Update" "Update the current row "
      if active_rec then
         call input_rec( appl_rec.*, "MODIFY" ) returning ⇐
appl_rec.*
         call upd_rec( appl_rec.* )
      else
            call err_msg( "You must Query for a record ⇐
first." )
      end if

   command "Delete" "Delete the current row"
      if active_rec then
         call delete_rec( appl_rec.* )
      else
         call err_msg( "You must Query for a record first." )
      end if

   command "Next" "Move to the next row"
      if active_rec then
         call curs_move( +1 ) returning retr_rec.*
         if retr_rec.acct_num IS NOT NULL then
            let appl_rec.* = retr_rec.*
            call display_rec( appl_rec.* )
         end if
      else
         call err_msg( "You must Query for a record first." )
      end if

   command "Previous" "Move to the previous row"
      if active_rec then
```

```
            call curs_move( -1 ) returning retr_rec.*
            if retr_rec.acct_num IS NOT NULL then
               let appl_rec.* = retr_rec.*
               call display_rec( appl_rec.* )
            end if
         else
            call err_msg( "You must Query for a record first." )
         end if

      command "Exit" "Exit the menu"
         exit menu

   end menu

end main
```

The qbe Function

The qbe function performs query by example. This function works on the assertion that the screen form is currently displayed. It will merely call the `construct` statement using the `construct by name` syntax and then return the resulting string to the calling function.

```
function qbe()
define qstr char(500)

construct by name qstr on accounts.*

return qstr clipped
end function
```

The open curs Function

The open_curs function is called after the qbe function has been called. This function will take the string containing the query and prepend it with the leading portion of the `select` statement. It will first `prepare` the statement and then check for errors. If any errors have occurred, the `err_handler` function will have been triggered by the `whenever error` statement in the main program block. The `err_handler` will have displayed an error message on the terminal screen, so there is no need to do that here. However, since there is no need to continue with the function if the `prepare` statement failed, the application record (`appl_rec`) is set to NULL and returned to the calling function.

If the `prepare` operation succeeded, a cursor is declared for the statement id and then opened. A `fetch` operation is then attempted on the cursor. If there is an error on the `fetch`, the application record is set to NULL and returned to the calling function. If there are no rows found for the `fetch`, an error message is displayed indicating this condition (since the `whenever error` statement will not trap this condition), and the application record is set to NULL and returned.

If all has gone well, the application record is simply returned to the calling function. The code for the `open_curs` function is shown below.

```
function open_curs( qstr )
define qstr char(500)
define appl_rec record like accounts.*

let qstr = "select * from accounts where ", qstr clipped
prepare st1 from qstr
if sqlca.sqlcode < 0  then
   initialize appl_rec.* to NULL
   return appl_rec.*
end if

declare appl_curs scroll cursor for st1

open appl_curs
fetch relative 1 appl_curs into appl_rec.*
if sqlca.sqlcode < 0  then
   initialize appl_rec.* to NULL
   return appl_rec.*
end if

if sqlca.sqlcode = NOTFOUND then
   error "No rows found."
   initialize appl_rec.* to NULL
end if

return appl_rec.*
end function
```

The `display_rec` Function

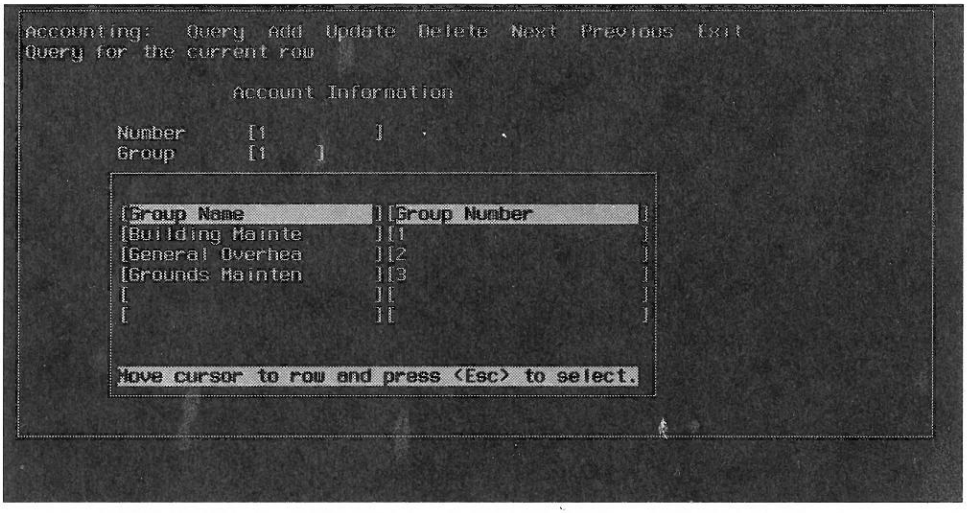
The `display_rec` function will display a record that has been passed to the screen. It does this by using the `display by name` form of the `display statement`. Once the data has been displayed, the function simply returns to the calling function.

```
function display_rec( appl_rec )
define appl_rec record like accounts.*

display by name appl_rec.*

end function
```

The input_rec Function

The input_rec function is used to allow the user to input data into the screen form. The input statement uses the input by name form of the input statement. This particular application performs no edit checks, though they could easily be added to the input block.

Figure 11-3 Template Application Zoom Function



When the user presses the Control-P key, the *zoom* key, a case statement is executed to determine the field where the cursor is currently resting. This version of the case statement does not specify a variable to be evaluated in each when clause, but instead evaluates an expression in each where clause. This is effectively an easier-to-read version of an if/then/else statement.

Depending on the field, the zoom2 function is called. The function is passed the name of the table, the fields from the table which should be displayed, the where clause criteria and then the labels for the two fields to be displayed (see Figure 11-3).

The zoom2 function will display a list containing the two columns that were passed into the function. It will only display a list that matches the where clause criteria

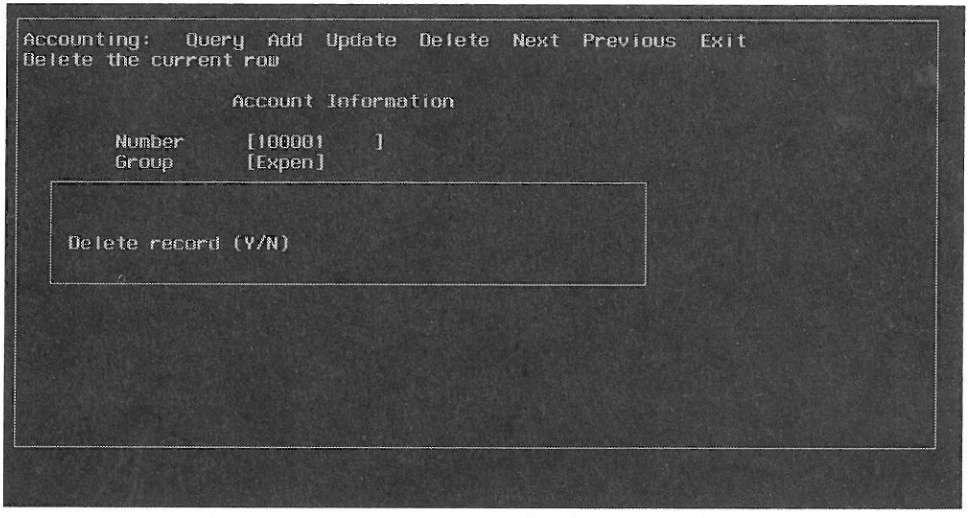
passed. It will return two values, representing the two array elements that were the current row when the user pressed the ACCEPT key. Since this application is only interested in one of these values, a dummy string variable is used to store the value; the other value will be stored in the appropriate element of the application record. Once the value has been returned, it is displayed to the appropriate field and the `case` statement is exited.

```
function input_rec( appl_rec, mode )
define appl_rec record like accounts.*
define mode char(10)
define str1 char(20)

if mode = "ADD" then

   initialize appl_rec.* to NULL

end if

input by name appl_rec.* without defaults

-- on key
on key (Control-p)

   case

     when infield( acct_group )
            call zoom2( "acct_groups", "group_name", "acct_group",
                        "1=1", "Group Name", "Group Number" )
                 returning str1, appl_rec.acct_group
            display by name appl_rec.acct_group

     when infield( acct_type )
            call zoom2( "acct_types", "type_name", "acct_type",
                        "1=1", "Type Name", "Type Number" )
                 returning str1, appl_rec.acct_type
            display by name appl_rec.acct_type

   end case

end input

return appl_rec.*
end function
```

The upd_rec and delete_rec Functions

The upd_rec function accepts an application record as a single parameter. It then takes a portion of the record and updates specific, nonkey fields of the target table for the application. In this case, three fields in the accounts table are updated from the application record.

Figure 11-4 Template Applicaton Delete Prompt



The delete_rec function receives an application record as a parameter. It will first prompt the user, using the prompt_msg library function presented in Chapter 14, to confirm the delete operation (see Figure 11-4). If the user responds with a Y, then, using the key fields from the application record, the function deletes the record in the table that corresponds to the application record. In this case, the acct_num field in the accounts table is the primary key for that table. Using this key, the delete statement deletes a single record.

```
function upd_rec( appl_rec )
define appl_rec record like accounts.*

update accounts
   set ( acct_group,acct_type,acct_name)  =
       ( appl_rec.acct_group, appl_rec.acct_type, appl_rec.acct_name )
   where accounts.acct_num = appl_rec.acct_num
end function

-----------------------------------
```

```
function delete_rec( appl_rec )
define appl_rec record like accounts.*
define resp char(1)

call prompt_msg( "Delete record (Y/N) " )
    returning resp

if upshift( resp ) = "Y"  then

  delete from accounts
     where acct_num = appl_rec.acct_num

end if


end function
```

The curs_move Function

The curs_move function is used to move forward and back in the current set. It receives a single parameter, an integer. This integer variable is used in the fetch relative statement to move the number of records specified in the direction variable. If the contents of the direction variable is a positive one, the fetch relative statement will move forward one record in the active set. If the value is a negative one, the cursor will move back one record in the active set.

The fetch relative statement will retrieve one record into the application record and return the record. If a NOTFOUND condition is encountered, the application record will be set to NULL and returned.

The open_form and add_rec Functions

The open_form function will simply open the screen form and display it to the screen. The add_rec function takes a single record, which represents the record to be inserted. It then inserts that record into the database. Error checking will be trapped by the whenever error statement.

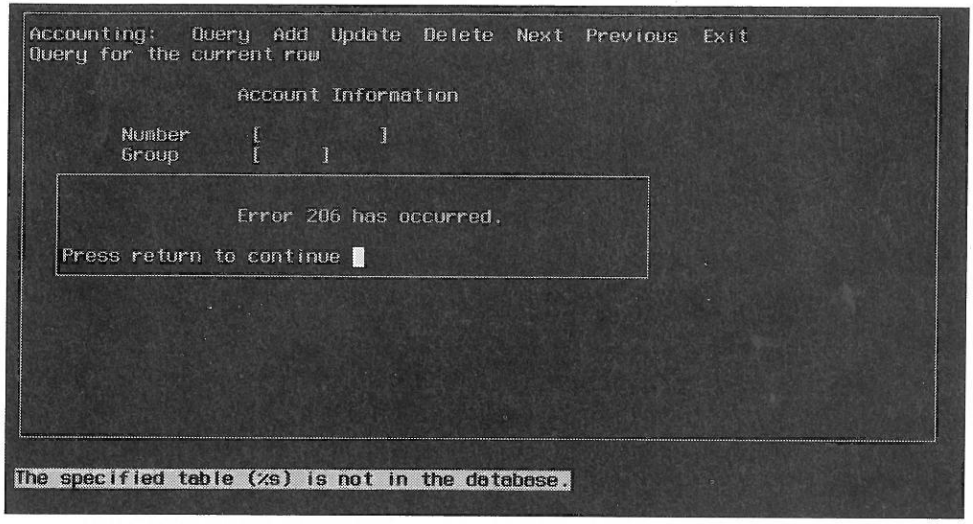
```
function curs_move( direction )
define direction smallint
define appl_rec record like accounts.*

fetch relative direction appl_curs into appl_rec.*

if sqlca.sqlcode = NOTFOUND then
   error "No more rows."
```

```
end if

return appl_rec.*

end function

------------------------------

function open_form()

open form appl_form from "accounts"
display form appl_form

end function

------------------------------

function add_rec( appl_rec )
define appl_rec record like accounts.*


insert into accounts values ( appl_rec.* )


end function
```

The `err_handler` Function

The `err_handler` function is used to process errors trapped by the `whenever error` statement. It will print an error message on the error line in reverse video for the SQL error code that has occurred. It will also make use of the `err_msg` function, shown in Chapter 14, to display the error message to the screen in a window and prompt the user to press return. Once these two tasks are complete, the function will return to the point in the application code where the error occurred. Figure 11-5 illustrates the error-handling return prompt.

Figure 11-5 Template Application Error Handling


Accounting: Query Add Update Delete Next Previous Exit
Query for the current row
Account Information
Number [ ]
Group [ ]
Error 206 has occurred.
Press return to continue
The specified table (%s) is not in the database.

```
function err_handler()
define msgstr char(60)
let msgstr = sqlca.sqlcode using "Error <<<< has occurred."
call err_print( sqlca.sqlcode )
call err_msg( msgstr )

end function
```

CONVERTING THE TEMPLATE APPLICATION TO PROCESS OTHER TABLES

One advantage of using a template is to provide similar functionality for a number of applications. Since the functionality will remain the same, the menu statement and the flow of control will not change from application to application. What will change are the names of the table or tables and the key fields in the tables.

In order to enhance modularity, the application record is declared local to the main module, and is then passed in and out of the various functions that are used to process the record. In order to change the structure of the application record, all definitions would have to be changed. The declaration for this record appears once in the main program module. The record is then declared as a parameter in a number of other functions. Since the naming convention for the application record is consistent throughout the application, a search for the pattern "appl_rec" would find all occurrences of the application record definition.

The screen form is also bound to the database table or tables. This must be changed to support the new table or tables, and the name of the screen form must be changed in the `open_form` function.

Additional field validations and table pick lists may also be used. These would be added as needed to the `input_rec` function within the input block.

SUMMARY

Using template applications can help maintain consistency and enhance the modularity of an application. This chapter presented only an input application, but reports could also use a similar template structure.

Templates will also help shorten the learning curve for the maintenance programmer. If all applications use the same flow of control and call the same functions, the maintenance programmer will have an easier time finding what they need within the application.

However, templates should not be a means of constraining the programmer. Not all applications will fit within the confines a few templates. A multiscreen application, which requires coordination between the data entered on several screens and includes an input array statement as part of a master detail relationship, would not fit neatly into the application just presented. As always, the application development process must be fluid. The tools and methods used to develop each application must be weighed against the complexity of the application, the length of time allowed to develop the application and the experience of the programmer. A wide range of possibilities exist, from developing the application from the ground up and not using any template code to developing the application within a strict set of guidelines using very specific templates; it is up to the programmer or project leader to make the correct choice.

Chapter 12

Optimizing INFORMIX-4GL Applications

One of the major advantages of INFORMIX-4GL over many other 4GLs is that it executes quickly. This is true of both the p-code version and the C language-compiled version. In many cases, the performance of the p-code version is more than adequate. In fact, this is true of most data entry applications and some reports. However, there may be applications where absolute peak performance is necessary. This could be the case with large reports, which must process many rows, batch processing applications, which must also read and process numerous rows, and some data entry applications. All of these applications may benefit from the speed of compiled INFORMIX-4GL.

Database-Bound Applications

In applications that are *database-bound*, spending most of their effort retrieving or writing data to the database, the speed of the front-end application language may not have significant impact. With these applications, most of the work will be performed by the database engine or back end, and improving the performance of the front-end application will have minimal, if any, impact on overall performance. An example of such an application is shown below. This application allows the user to input a small amount of data and then updates the database.

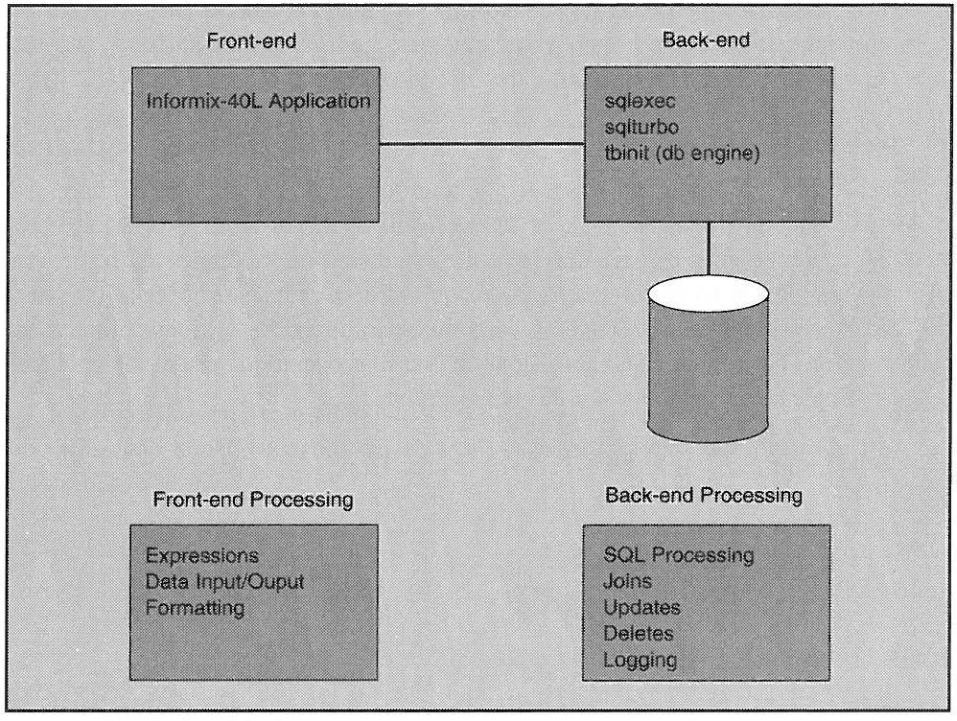
```
main

define old_acct_number, new_acct_number like accounts.acct_number
define resp char(1)

prompt "Enter old account number: " for old_acct_number
prompt "Enter new account number: " for new_acct_number

message "Please wait ... "

begin work

call change_acct_number( old_account_number, new_account_number )

message "Update complete. "

prompt "Commit work (Y/N): " for char resp
```

```
if upshift(resp) = "Y" then
   commit work
else
   rollback work
end if

end main

--------------------------------------------------------------------
function change_acct_number( old_acct, new_acct )
define old_acct, new_acct like accounts.acct_number

update accounts
set    acct_number = new_acct
where  acct_number = old_acct

update journals
set    acct_number = new_acct
where  acct_number = old_acct

update acct_totals
set    acct_number = new_acct
where  acct_number = old_acct

end function
```

In this example, the user enters the old account number and the new account number. A function is then called to change the account number in the database by updating the various tables that contain the account number. The user is then given the choice of committing the update or rolling it back.

In Figure 12-1, the front-end application performs minimal work. It provides some screen interaction with the prompt and message statements, but then executes SQL statements to perform the updates. All SQL statements are sent directly to the database engine to be executed; the front-end will call a function, which will pass a string containing the SQL statement to the engine via an Inter Process Communication (IPC) facility. The front-end application will then wait while the database engine performs the work. When the database engine is finished, it will send a message back to the front end, and the front end will continue its work. The update of the tables and any logging used to allow rollbacks are performed by the database engine; the front-end application merely passes the SQL statements and receives information on the success or failure of the operation.

Figure 12-1 Front-End versus Back-End Processing



The majority of work performed in executing this application is performed at the database level. The front-end work on retrieving data is trivial compared to the expense of updating the database tables with the new account numbers. This application qualifies as a *database-bound* application. If the number of rows to update is large, then the database operation could take some time. While the update is being performed, the front-end INFORMIX-4GL application will be waiting for the engine to return a confirmation that the statement executed correctly.

To the user running this application, it may appear that the program is running slowly. In fact, it is the speed of processing in the database engine that will most heavily impact the speed of this application. Optimizing the execution of the SQL statement would be the best way to improve the speed of the application. This optimization could involve examining the tables to see that index access is available and, if the database is Informix-Online, tuning the database engine to ensure that the updates are being processed efficiently by the engine.

Identifying Repetitive Code

In optimizing any application, code should be reviewed to determine which sections will be repeated most often. This will help identify code that can be optimized.

If code is executed repeatedly, any expensive operations incur a cost equal to the cost of the operation times the number of times the code is executed. The larger the number of iterations, the more significant the cost of the operation.

Decision Operations

Decision operations incur a penalty; a stack operation is required to place the decision operand on the stack. If 1 of the decision operands is an expression, then expression operation must be executed since the expression must be resolved before the decision operation can be performed. If the section of code is repeated several thousand times, the expression must be resolved several thousand times. An obvious optimization would therefore be to eliminate expressions in decision operations whenever possible. The code sample below demonstrates this optimization.

```
foreach curs1 into rec.*

if ( x + y - z ) then
   call get_data( rec.col1 ) returning rec1.*
end if

end foreach
```

Since the expression within the decision operation is not dependent on the data retrieved in the `foreach` loop, the expression need not be performed on each iteration of the loop. Performing the expression outside of the loop and storing the result is more efficient, as shown below.

```
let res = x + y - z

foreach curs1 into rec.*

if ( res ) then
   call get_data( rec.col1 ) returning rec1.*
end if

end foreach
```

This code eliminates the need to evaluate the expression on each pass. If this `foreach` loop were reading 1,000,000 records, and each expression evaluation took 20

milliseconds, this could produce a savings of approximately 30 minutes in the runtime of the application.

Optimizing INFORMIX-4GL Loop Operations

The INFORMIX-4GL looping operations do not always perform expression evaluation when expected. In the `for` loop, if an expression is used in the initialization, it is evaluated on *each* pass of the loop to allow for variable modification within the loop. An example is shown below.

```
for x = z + y / 2 to 1200

    call fix_data( rec.* )

end for
```

The initialization expression in this `for` loop will be evaluated on each pass of the loop. This loop could be optimized by removing the expression evaluation from the initialization as follows

```
let res = z + y / 2

for z = res to 1200

    call fix_data( rec.* )

end for
```

The same expression evaluation rules hold true for the other parameters used in the `for` loop: the limit parameter and the step parameter. They could also be optimized as shown below.

```
let limit    = (10 * x ) * z
let step_amt = i + 4
let start    = 10 + x

for n = start to limit step step_amt

    call retrieve_data( rec.* ) returning rec1.*

end for
```

OPTIMIZING FUNCTION CALLS

All function calls in INFORMIX-4GL require stack operations and, as in most other languages, these stack operations incur a cost. Reducing these stack operations can help optimize an application. Since modularization encourages parameter passing, the elimination of all parameters is discouraged. What is encouraged is the elimination of unnecessary or redundant parameters.

In INFORMIX-4GL, the use of the `record.*` syntax makes it easy to pass an entire record to a function. However, this often leads to unnecessary parameter passing, as shown below.

```
call retrieve_rec( rec1.* ) returning rec2.*

...

function retrieve_rec( rec.* )
define rec record like table2.*
define ret_val    like table1.col1

if  rec.col1 = "H" then
  select col1
  into   ret_val
  from   table1
  where  table1.col1 = rec.col2
end if

return ret_val

end function
```

In this example, the function `retrieve_rec` receives what appears to be a single parameter, a record of `table2` data. In INFORMIX-4GL, however, because all function calls are *call-by-value*, this parameter results in all of the elements of the `table2` record being placed onto the stack. If `table2` has a large number of elements, this could be an expensive function call. In order to optimize this application, the number of parameters passed must be reduced, as shown below.

```
call retrieve_rec( rec1.col2 ) returning rec2.col1

...

function retrieve_rec( col2 )
```

```
define col2    like table2.col2
define ret_val like table1.col1

if  key1 = "H" then
  select  col1
  into    ret_val
  from    table1
  where   table1.col1 = col2
end if

return ret_val

end function
```

In this example, only the parameters required by the function are passed into the function. If `table2` contained 10 columns, this would eliminate 9 stack operations that would have had to be performed if the entire `table2` record had been passed.

Eliminating Unnecessary Statements and Expressions

Complex expressions and statements require CPU time to complete. When executed many times, this expense can add up; but obviously not all complex expressions can be eliminated. The expressions to identify are those that will be executed many times and should be reduced or eliminated whenever possible.

Often, these expressions are contained within a loop. Expressions to eliminate are those that are not dependent upon values created within the loop; these expressions can be evaluated before entering the loop and stored in a variable. This variable can then be interrogated within the loop instead of executing the expression on each loop iteration, as shown in the following code.

```
foreach curs1 into rec1.*

    if ( ( x * y )- z ) = rec1.col2   then

        let rec1.col3 = str1[3,10]
        let rec1.col4 = (4 * rec1.col5 )/ 100

    end if

     call update_rec( rec1.* )

end foreach
```

In this example, there are several expressions within the `foreach` loop that are not dependent on values retrieved within the loop. The first is the expression within the `if/then` statement. The next is the substring operation in the assignment for the `rec1.col3` record element. The last is the mathematical expression that assigns a value to the `rec1.col4` record element. However, this expression is based on the `rec1.col5` field that is retrieved on each pass of the loop; this expression cannot be optimized since it is dependent on values retrieved on each loop iteration. Only the string expression and the first mathematical expression can be optimized, as is shown below.

```
define expr1 smallint
define sstr char(8)

let expr1 = ( x * y ) - z
let sstr  = str1[3,10]

foreach curs1 into rec1.*

     if expr1 = rec1.col2   then

         let rec1.col3 = sstr
         let rec1.col4 = ( 4 * rec1.col5 ) / 100

    end if

     call update_rec( rec1.* )

end foreach
```

By placing the expressions into variables, the cost of evaluating the expressions on each pass is eliminated. If the expression evaluation cost .002 CPU seconds and the `foreach` loop were reading 2,000,000 records to perform its update operation, this optimization would save approximately 60 minutes in processing time.

The final optimization in this application would be the elimination of unnecessary statements. In this example, a function call is made to update the record that has been modified. If the update function is small and merely updates 1 or two tables, it may be desirable to place the code for the update *in-line,* placing it within the `foreach` loop. This would avoid the stack operations involved in the function call.

Reducing String Operations

In INFORMIX-4GL, function parameters are passed *by value*. When a character string is passed to a function in INFORMIX-4GL, all the bytes associated with the string

are popped onto the stack. On returning a string from a function, the string is once again placed onto the stack. Often, even when the `clipped` operator is specified, the entire defined length of the string is pushed and popped from the stack.

The best optimization may appear to be declaring strings to be globals and eliminating the need to pass them as parameters, but this runs counter to the general rules of modularization where parameter passing is encouraged. A more appropriate solution for a modular application would be to review the parameters being passed into a function. If a multibyte character string flag is being passed, it may be better to use a single-byte character string for the flag or to use a 2-byte integer (a `smallint`) for the flag parameter. The following code demonstrates this optimization.

```
define mode, flag1, flag2, flag3 char(10)

call update_db( mode, flag1, flag2, flag3, rec.* )

...

function update_db( mode, flag1, flag2, flag3,  rec )
define mode, flag1, flag2, flag3 char(10)
define rec record like table1.col3 thru table1.col7


if mode  = "UPDATE" and
   flag1 = "ACCOUNT_MTH" and
   flag2 = "JOURNALS"    and
   flag3 = "UPDATE1"       then

  update table1
  set (col3, col4, col5, col6, col7 ) =
      (rec.col3, rec.col4, rec.col5, rec.col6, rec.col7 )

end if

end function
```

In this example, the mode and several other flags are passed into the function as 10-byte character strings. Reducing these parameters to 1-byte character strings, as shown below, eliminates 36 bytes being placed onto the stack.

```
define mode, flag1, flag2, flag3 char(1)

...
```

```
call update_db( mode, flag1, flag2, flag3, rec.* )

...

function update_db( mode, flag1, flag2, flag3,  rec )
define  mode, flag1, flag2, flag3 char(1)
define rec record like
      table1.col3 thru table1.col7

if mode  = "U" and
   flag1 = "M" and
   flag2 = "J" and
   flag3 = "1"        then

  update table1
  set  (col3, col4, col5, col6, col7 ) =
       (rec.col3, rec.col4, rec.col5, rec.col6, rec.col7)

end if

end function
```

The 10-byte character string flags used in the previous example are replaced by single-byte parameters, thus reducing the number of bytes placed on the stack on each pass. However, this has the impact of reducing the readability of the application to a certain extent. This issue can be addressed by adding global *constants* to the application to improve the readability, as shown below.

```
define mode, flag1, flag2, flag3 char(1)
define UPDATE_MODE, MODIFY, JOURNALS, FIRST char(1)

...
-- set constant values
let UPDATE_MODE   = "U"
let MODIFY        = "M"
let JOURNALS      = "J"
let FIRST         = "1"

call update_db( mode, flag1, flag2, flag3, rec.* )

...

function update_db( mode, flag1, flag2, flag3,  rec )
```

```
define mode, flag1, flag2, flag3 char(10)
define rec record like table1.col3 thru table1.col7


if mode  = UPDATE_MODE and
   flag1 = MODIFY      and
   flag2 = JOURNALS    and
   flag3 = FIRST        then

  update table1
  set  (col3, col4, col5, col6, col7 ) =
       (rec.col3, rec.col4, rec.col5, rec.col6, rec.col7)

end if

end function
```

In this example, the advantage of shorter parameters is maintained while adding readability using *constants*. Of course, the current versions of INFORMIX-4GL do not support true constants; these variables are only constants if the programmer makes them so.

ELIMINATING PARAMETERS

Since every parameter passed into a function requires two stack operations, 1 to place the parameter on the stack and 1 to remove it from the stack, eliminating some parameter passing will help reduce stack operations. This requires records to be declared with module or global scope. Such a change reduces modularity, but is an effective optimization and a very common practice in languages that do not support *call-by-reference*.

In 1 of the examples shown above, the function received a record parameter as follows.

```
function update_db( mode, flag1, flag2, flag3,  rec )
define mode, flag1, flag2, flag3 char(10)
define rec record like table1.col3 thru table1.col7
```

This parameter requires 7 stack operations to place the referenced record components on the stack. By using a global variable in place of the parameter, the 7 stack operations are eliminated entirely, as shown below.

```
globals
   define rec like table1.col3 thru table1.col7
```

```
end globals

...

function update_db( mode, flag1, flag2, flag3 )
define mode, flag1, flag2, flag3 char(10)

...
```

SQL Optimization

The total spectrum of SQL optimization is beyond the scope of this chapter. Many applications, from data entry to batch processing and reporting, can benefit from this type of optimization. INFORMIX-4GL processes SQL statements synchronously, waiting for the SQL statement to finish before continuing processing. If a particular SQL statement were to process slowly, the INFORMIX-4GL application would also appear to run slowly. SQL statement execution often represents the *bottleneck* in an INFORMIX-4GL application.

When an SQL statement is executed, several events take place in the front and back ends. In order to process the SQL statement, it must be parsed, interpreted and executed. During the parsing and interpretation of the statement, database objects are evaluated for their existence, and some binding takes place between the front-end host variables and the database elements they reference. Memory is allocated in the front end to accept the data, and engine resources are allocated to process the SQL request.

There is a certain amount of overhead involved in these steps. By using the `prepare` statement, some of the overhead can be avoided on successive execution of the statement. This implies, of course, that the statement will be executed multiple times. If this is not the case, and the statement will only be executed once, there would be no benefit to preparing and then executing the SQL statement. Chapter 3 covers the `prepare` statement in more detail and provides examples of its use. Using the `prepare` statement judiciously can provide *significant* optimization; its use should not be overlooked when optimizing.

Summary

Optimizing applications is dependent upon the needs of the application. If the application is designed to perform data entry and update the database after each screen, optimization will have minimal impact on the speed of the application. If the application is a report that will read and process a million or more records, optimization could definitely impact the execution time of the report.

Optimization often requires striking a balance between maintaining the modularity of an application and improving the execution speed by reducing expensive operations. Some elements of modularity, such as parameter passing, require additional processing for the application. Reducing these instructions can optimize an application, but leads to erosion of the application's modularity. The programmer must often make the decision: optimize the application and save 10 minutes of processing time per run, or leave the application as is, maintain modularity and save hours in program maintenance time.

Chapter 13

An INFORMIX-4GL Development Environment

An INFORMIX-4GL development environment can make use of the development tools provided by Informix; alternatively, development can be done in the UNIX environment making use of the UNIX tools provided for program compilation and source code control. Using the UNIX tools requires only minor changes to create a development environment similar to that of C code development under UNIX.

LIBRARIES

A cornerstone of any development environment is a library of INFORMIX-4GL routines. Libraries should be composed and organized before development begins, and then maintained and appended as development continues. A general set of library routines, such as those presented in Chapter 14, should be included as well as other, more application-specific libraries. For instance, an application that is developing an accounting application could contain such libraries as those listed in Table 13-1.

Table 13-1 Sample Development Environment

Library	Purpose
libgen.4gl	General-use routines—string manipulation, sort, prompts, error messages
libscreen.4gl	Screen-handling routines—pick list, zoom windows
libreport.4gl	Report routines—report header, trailer
libacctg.4gl	Accounting-specific functions—depreciation routines, account totals, account rollup
libfin.4gl	Financial routines—present value, future value

Such a division of library routines serves two purposes. First, it makes maintenance of the libraries easier since functions are logically divided into separate components. Second, it makes inclusion of the routines more efficient for the p-code version of INFORMIX-4GL, which does not select specific routines from a *linked* module, but

instead links the entire module to the other source code modules by concatenating the files together.

UNIX MAKEFILES

The UNIX `make` utility (also readily available in the DOS environment) provides an excellent tool for controlling and automating the program compilation process. With some minor modifications, it can be made to recognize the differences between the compilation of p-code INFORMIX-4GL and compiled C-code INFORMIX-4GL.

By default, the `make` utility assumes a certain set of file name suffixes exist. The `make` utility uses this suffix list internally to make decisions about compiling the target application. In the case of compiling a C code application, `make` would first read the *makefile*, the list of rules and objects needed to create the target application. The `make` utility would first determine the objects needed to create the application. It would then search for files with an `.o` suffix and a `.c` suffix. If the file with a `.c` suffix had been touched more recently than the file with an `.o` suffix, the `.c` file would be recompiled using the default compiler, `cc`.

This set of compile rules does not match the needs of an INFORMIX-4GL application since the INFORMIX-4GL suffix rules and compilers are different, but fortunately, `make` allows its compile rules to be redefined. Three elements of the `make` compile rules need to be changed: the suffix list, the method for compiling objects, and the method for creating the target application. These could be redefined on a file-by-file basis in the makefile, but it is more convenient to simply inform `make` of the suffix rule changes in each makefile used. The makefile file below demonstrates the method for doing this.

```
.SUFFIXES:
.SUFFIXES: .4gl .ec .4go .o .4ge .4gi .per .frm

OBJECTS=gen_tab.4go gen_main.4go gen_input.4go
FORMS=gen_tab.frm
TARGET=gen_tab.4gi
LIBS = libgen.4go

C4GL=c4gl
FGLPC=fglpc
FORMBLD=sformbld
LINKER=cat

$(TARGET): $(OBJECTS) $(FORMS)
  $(LINKER) $(OBJECTS) $(LIBS) > $(TARGET)

.4gl.4go:
  $(FGLPC) $<
```

```
.4gl.o:
  $(C4GL) -c $<
.per.frm:
  $(FORMBLD) -s $<
```

The first line of the makefile resets the suffix rules for the `make` utility. If this line had just read

```
.SUFFIXES: .4gl .ec .4go .o .4gi
```

the `make` utility would have *appended* these suffixes to its internal suffix list. Since the goal is to replace the default suffix list, the first line resets the suffix list, and the next line indicates the order of the suffixes for INFORMIX-4GL compiles.

Next, a series of `make` variables are declared. These variables will be used later in the file to define compilation rules for this application. Declaring variables in a makefile makes the file more readable and makes modification easier. Rather than search the entire file for a module name, it could simply be changed in this location, and the change would be recognized throughout the file.

A variable is declared for OBJECTS. The files listed on this line are the object files needed to create the application. A variable is also declared for FORMS; the contents of this variable will indicate the screen form or forms that are needed to run the application. The TARGET variable contains the name of the application to be made. The LIBS variable lists the libraries that are needed to compile the application.

Variables are then declared to indicate the programs needed to compile the application. Declaring these as variables makes it easier to compile the application in an environment where there may be more than one version of INFORMIX-4GL in use, each using different compilers.

Once all of the variables have been declared, the first line of the rules section is the first line that the `make` utility will act on if no specific targets have been specified. This line contains only variable names, once again making modification of the makefile easier.

The first parameter on this line is the target to be created using the rules specified. In this case, the target is the contents of the variable named, appropriately enough, TARGET. Following the colon is a list of objects required to create the target. The contents of the variable OBJECTS resolves to a string containing the object modules used to create the target application. The screen forms are also specified on this line in the FORMS variable, thus forcing `make` to create these before creating the target application. The LIBS variable is not specified on this line, thus leaving the creation of the library module to some other makefile.

The next line specifies the rule for creating the target application. This line must be preceded by a TAB character. It will be executed by `make` only when it has a current copy of the objects listed on the previous line. Once `make` has all of these objects

available and current, it will create the application using the rule specified on this line. In this case, the linker will be used to take the object modules and libraries specified in the LIBS variable and create the target application. Since the LINKER variable has been declared as the UNIX `cat` utility, the p-code object files specified in the OBJECTS variable will merely be concatenated together with the libraries specified in the LIBS file to create the target specified in the TARGET variable. Note that the FORMS variable specified on the previous line is not used on this line. The `make` utility does not require that all of the objects specified be used on the rules lines.

The remaining lines of the makefile specify rules for creating a file with one suffix from a source file with a different suffix. The syntax is as follows.

```
.suffix_to_change.new_suffix:
   <rule for creating new suffix>
```

The first element on this line is the suffix to change, preceded by a period. The next element is the suffix to create. This rule will be invoked as `make` scans the list of objects needed to create its target. When this rule is invoked, `make` will have an object available, so there is no need to specify an object on the objects line.

On the rules line, the rule for creating the object is listed. In this case, the program to be invoked to create the object is specified followed by the `make` utilities arcane syntax for the current object, a dollar sign adjacent to a less than sign ($<). As with other `make` rule lines, this line must be preceded by a tab.

The remaining lines of the makefile are used to specify additional rules: rules for creating a compiled screen form and a C-language-compiled object file.

Supporting P-code INFORMIX-4GL and Compiled C-code Versions with Make

It is not uncommon to support both p-code versions of an application for a development environment, and C-code versions for production environments. This strategy has the advantage of rapid compiles and good debugging tools with the p-code version of INFORMIX-4GL during the development process, and fast execution for the compiled C-code version in the production environment. If the makefile were to be used to support both p-code 4GL and C-language compiled 4GL, the following lines would be added to provide this support.

```
CTARGET=gen_tab.4ge
COBJECTS = gent_tab.o gen_main.o gen_input.o
CLIBS = libgen.o
....
```

```
$(CTARGET): $(COBJECTS) $(CLIBS) $(FORMS)
  $(C4GL) $(COBJECTS) $(CLIBS) -o $(CTARGET)
```

Lines are added to specify the name of the compiled C target file, the objects to be used to create the target file and the libraries to use when the objects are linked. A rules line is then added to specify how the target is to be created.

The COBJECTS variable indicates the source code object modules needed to create the target. The CLIBS variable indicates the libraries to use to create the target. So, as with the p-code target, the screen form used in the program is specified.

The following line specifies how to create the target. The linker, which in this case is the c4gl script, is passed parameters for the objects and libraries. The -o flag is used to specify the name of the output file, which is the contents of the CTARGET variable.

Source Code Control Support in Makefiles

Adding support for source code control in a makefile is a relatively easy task. The rule specifying how to compile the file is modified to first retrieve the file from the source code control system as follows.

```
.SUFFIXES:
.SUFFIXES: .4gl .ec .4go.o .4ge .4gi .per .frm

OBJECTS=gen_tab.4go gen_main.4go gen_input.4go
FORMS=gen_tab.frm
TARGET=gen_tab.4gi
LIBS = libgen.4go

CO=co
#CO=sccs get
C4GL=c4gl
FGLPC=fglpc
FORMBLD=sformbld
LINKER=cat

$(TARGET): $(OBJECTS) $(FORMS)
  $(LINKER) $(OBJECTS) $(LIBS) > $(TARGET)

.4gl.4go:
  $(CO) $<
  $(FGLPC) $<
.4gl.o:
  $(CO) $<
  $(C4GL) -c $<
```

```
.per.frm:
  $(CO) $<
  $(FORMBLD) -s $<
```

A line has been added to the variable definitions to define the utility used to retrieve files from the source code control system. In this example, the UNIX `co` utility is used to retrieve the file. This source code control command does not retrieve an editable copy of the file; it is up to the programmer to do that. (Obviously, more complex controls could be placed on the source code than those shown here.)

A line has then been added to the rules lines for the compilation of the objects. This line will retrieve the file from the source code control system. The next line of the rule for creating the target is the same as in the previous example; it compiles the current source to create the object currently being processed.

Recognizing Directories in a Makefile

Often, the directory in which the source code, forms or libraries are located is not the programmer's current directory. Declaring a variable to indicate where these files are located can make maintaining the makefile easier. If the files are moved to another directory, a quick change to the makefile will recognize the change, as shown below.

```
LIBDIR=/usr/src/informix/acctg/libs
LIBS=$(LIBDIR)/libacctg.4go $(LIBDIR)/libfin.4go

...

$(TARGET): $(OBJECTS) $(FORMS)
      $(LINKER) $(OBJECTS) $(LIBS) > $(TARGET)
```

Summary

By making use of some very simple facilities, a modular development environment can be maintained. With some simple modifications, many of the development utilities that make UNIX such an excellent development environment can be applied to INFORMIX-4GL applications.

Chapter 14

General Function Library Routines

This chapter presents a set of general-use library routines that are used throughout this text. Some of the routines mirror the capabilities of general library functions in other languages such as C or Basic; others are peculiar to the needs of INFORMIX-4GL programming. Their use, as with any set of library routines, can enhance the modularity of an application and thus improve the overall productivity of application development with INFORMIX-4GL. Table 14-1 lists the functions, their parameters and their purpose.

Table 14-1 General Function Library

Function Name / Parameters	Purpose
`err_msg( msg_str )`	To open window and display an error message to the user. The user is then prompted for a key press. Returns no values.
`center( str, len )`	To center a string value parameter passed to the function (str) on length len. Returns the string passed in centered in the return string.
`all( char1, len )`	To duplicate the character passed into the function (char1) for len parameter number of times.
`prompt_msg( msg_str )`	To open window and display a prompt message to the user. The user is then prompted for a key press. Returns no values.
`is_number( str )`	Examines the string str for non-numeric characters. Returns TRUE if one was found.
`is_print_ascii( pchar )`	Examines the character passed to determine whether or not it is a printable ASCII character.
`find_str( search_pattern, string_buffer )`	Searches the search string passed for the search pattern passed. Returns the offset where the string was found, if found. If the search string is not found, the function returns FALSE.

continued on next page

Table 14-1 continued

Function Name / Parameters	Purpose
`replace_str( search_pattern, replace_pattern, string_buffer )`	Searches the string buffer for the search pattern. If the search pattern is found, it is replaced with the replace_pattern string passed to the function. The function returns the modified string buffer to the calling function.
`bsort( number_of_elements )`	Will sort the global array of strings for the number of elements passed. Returns no values, but leaves the values of the global array sorted in ascending order.
`soundex( str_name )`	Converts the character string passed into the function to a four-byte string. Returns the four-byte string.

The `err_msg` Function

The `err_msg` function displays a box on the screen and prompts the user to press return to continue. Since this is a modal-type dialogue box, it has the effect of forcing the user to read the error message before continuing with the application.

The function receives a single parameter, the error message to display. It takes the error message and centers it in the 80-character string passed into the function.

It then opens a window with four rows and 50 columns (obviously limiting error messages to under 50 characters in length). It displays the error message, using the bold attribute, and prompts the user to press return. Since the `prompt` statement is executed with the `for char` clause, the input of a single character will be adequate to end this function. The character typed is not checked; it can be the return character or any other character. Once finished, the function returns control to the calling function.

```
function err_msg( msg )
define msg     char(50)
define c1      char(1)

let msg = center( msg, 50 )

open window _ew1 at 10, 5 with 4 rows, 50 columns
          attribute( border, prompt line last )
```

```
display msg clipped at 2,2 attribute ( bold )
prompt "Press return to continue " for char c1

close window _ew1

end function
```

The center Function

The `center` function is used to center a string or pattern in the string to be returned. This is a useful function for centering headers or messages within column boundaries.

The function receives two parameters: a character string to center, currently limited to 80 characters; and a length parameter, indicating the length in columns on which to center the string.

The function gets the clipped length of the string passed and then uses this length in combination with the length parameter passed into the function to perform a calculation. The calculation that determines the column offset at which the centered string should be placed is as follows.

```
let x = ( len - l1 ) / 2
```

In this calculation, `len` is the length passed into the function; this is the length on which the string pattern should be centered. The `l1` variable represents the length of the string pattern passed into the function; this is the string that must be centered on the length `len`. The difference between `len` and `l1` is the difference between the length on which to center the string and the length of the string to be centered. This represents the amount of *space* or blanks that will be left when this string is displayed within the `len` length boundaries. Since it is our goal to center this string, placing half of the spaces to the left of the string and half of the spaces or blanks to the right of the string will result in a centered string. By dividing the difference between `len` and `l1` by two, we are halving the result of the operation and determining the proper column offset for the string pattern.

A `for` loop is then executed to pad the return string with blanks up to the column offset determined by the previous calculation. Once the return string has been padded, the string passed into the function is concatenated onto this string, thus placing the string at the correct centered offset within the length parameter passed into the function. The result of this concatentation operation is then returned to the calling function.

```
function center( str, len )
define str     char(80)
define len     smallint
```

```
define retstr char(80)
define n,x,l1 smallint

let l1 = length( str clipped )
let x = ( len - l1 ) / 2

for n = 1 to x
   let retstr[n] = " "
end for

let retstr = retstr[1,n], str clipped

return retstr

end function
```

The `all` Function

The `all` function takes two parameters: the character to be duplicated and the number of columns or times to duplicate the character. The function will duplicate the character the number of columns specified and then return the resulting string. This is a useful means of creating a string of characters to create a border line on an input screen or report header. It could also be used to initialize a string to blanks, an operation that is sometimes necessary when using the compiled C version of INFORMIX-4GL.

The function defines a local return string and then executes a `for` loop. For each iteration of the `for` loop, a single-character element of the return string is assigned the character parameter passed into the function. The `for` loop continues execution up to the length or columns parameter passed into the function. The resulting string is then returned to the calling function.

```
function all( ch, cols )
define ch char(1)
define cols,n smallint
define retstr char(80)

for n = 1 to cols
  let retstr[n] = ch
end for

return retstr

end function
```

The `prompt_msg` Function

The `prompt_msg` function displays a message to the screen and then prompts the user to press return to continue. This function is designed to provide a modal, window-based version of the `prompt` statement; it extends the `prompt` statement to add these capabilities.

The function opens a window with four rows and 50 columns. The `prompt` statement is then executed within this window. When the user has responded to the `prompt` statement, the window is closed, and the string value returned by the `prompt` statement is returned to the calling function.

```
function prompt_msg( msg )
define msg    char(60)
define retstr char(60)

open window _mw1 at 10, 5 with 4 rows, 50 columns
     attribute( border, prompt line last - 1 )

prompt " ",msg clipped," " for retstr

close window _mw1

return retstr

end function
```

The Validation Functions: `is_number` and `is_print_ascii`

The functions `is_number` and `is_print_ascii` perform validations on ASCII strings. The `is_number` function evaluates a string and returns TRUE if the string is a number and FALSE if the string is not a number. The `is_print_ascii` function evaluates a single character within a range of ASCII values; if the string does not fall within those values, the function returns FALSE.

The `is_number` function first determines the length of the string passed into the function. It then executes a `for` loop for the length of the string. Each element of the string is evaluated within a range of ASCII characters; if the character is *not* within this range of characters and is *not* a decimal or period character, the function will immediately return FALSE. If the function successfully executes the entire loop, the string is assumed to be a number and the function returns TRUE.

The `is_print_ascii` function takes a single character as a parameter. This character is evaluated as falling within a range of valid ASCII characters. If the character does not fall within this range, the function returns FALSE. If the character does fall within this range, the function returns TRUE.

```
function is_number( str )
define str char(60)
define n,len smallint
define c char(1)

let len = length( str clipped )

for n = 1 to len
   if NOT ( str[n] >= "0" and str[n] <= "9" ) then
       if ( str[n] != "." ) then
           return FALSE
       end if
   end if

end for

return TRUE

end function

-----------------------------------------------------------

function is_print_ascii( pchar )
define pchar char(1)

if pchar > "!" and pchar < "~"  then
   return TRUE
else
   return FALSE
end if

end function
```

The `find_str` Function

The `find_str` function searches a string for a pattern and returns the starting position of the string pattern. This is a useful function for scanning an input string to determine whether or not a user entered a particular pattern.

This function receives two parameters: the string pattern for which to search and the string to search. Both parameters are declared to be 100 characters in length. Several integer variables are declared to hold string lengths and offsets.

Next, the length of the two parameters is stored in variables. A `for` loop is then started, which will be used to scan the entire string to be searched. For each iteration of

the `for` loop, a starting byte is stored; this is the current offset into the string to be searched as stored in the `for` loop counter variable. The length of the string pattern being searched for is added to this value. These two values are then used to obtain a substring into the string being searched. This substring returns a string equal in length to the string pattern being sought. A series of these substrings, taken incrementally from the string being searched, are used to scan the string being searched for the search pattern. The `if/then` statement determines if the string pattern is present as shown below.

```
if find_str clipped = str[ x, y ] then     -- pattern has ⇐
    been found
....
```

If the string pattern is found, the function returns immediately at that point. In order to find successive occurrences of the string, the following approach could be used.

```
let x = TRUE
let len = length( str clipped )
while ( x )
      call find_str( "this", str[ x, ( len - x ) ] ) returning
            str, x
      if x then
          let found_count = found_count + 1
      end if

end while
```

This code would continue the `while` loop till the value of `x`, the position where the string was found, was 0. As long as `x` is returned as non-0, the loop will continue to execute. When no occurrences of the string are found, the value of `x` will be returned as a zero and the `while` loop will terminate.

```
function find_str( find_str, str )
define find_str, str char(100)
define n, x, y, len_find, len_str smallint

let len_find = length( find_str clipped )
let len_str  = length( str clipped)

for n = 1 to len_str - len_find

    let x = n
    let y = n + len_find - 1
```

```
   if find_str clipped = str[ x, y ]  then      # found it
         return  ( x )
      end if
   end for
   return 0
   end function
```

The replace_str Function

The replace_str function takes as its parameters a string pattern to find, a string pattern with which to replace the pattern, and the string to search. It returns the resulting string and the position where the replace operation took place, or zero if nothing was replaced. It does not act globally; it only replaces a single occurrence of the find pattern. If a global replacement is needed, a loop demonstrated below is needed.

```
let x = true
while ( x )
   call replace_str( "formonly", "table1",  str ) returning str,x
end while
```

In this example, the while loop is executed as long as the value of x is non-zero. The replace_str function will continue to perform substitutions until no more occurrences of the string pattern 'formonly' are found. When no more occurrences are found, the return parameter for the position where the string was found will be zero; thus, the value of x will be zero and the while loop will terminate.

The replace_str function receives three parameters: the string pattern to find, the string pattern with which to replace the found string, and the string to search. All parameters are declared to be 100-byte character strings. Variables are declared to hold the length of the strings and to be used as counter variables in loops. First, the lengths of all three string parameters are determined. Then the found_at variable, which will be used to store the position at which the string pattern was found, is set to zero. If no string is found, this variable will not be reset and will thus return the value zero.

A for loop is started and is executed up to the length of the string to be searched parameter. In this way, the entire string will be searched or scanned until the search

pattern is found. For each iteration of the `for` loop, the offset into the string to search, which is the counter variable for the `for` loop, is stored in the variable `x`. Added to this value is the length of the string pattern being sought, with the result being stored in the variable `y`. Now the expression `str[ x, y ]` where `str` is the string being searched, evaluates to a substring the same length as the string pattern being sought. Since these substrings are taken for each byte of the string being scanned, the result is a set of correctly sized substrings which is examined for the entire string being searched. These substrings are evaluated against the string pattern sought to determine if a match has been found.

If a match is found, the found string is replaced with the string designated as a replacement, the position where the string was found is stored and the `for` loop is exited. How the substring is taken is dependent upon the current offset into the string. If the string being sought is found at the first byte, the target string or replacement string is concatenated with the remainder of the string being searched. Otherwise, a substring of the string being searched is taken from the first byte to one byte less than where the string was found. This substring is concatenated onto the string designated as a replacement, and a substring of the string being searched from the end byte of the occurrence of the string being sought is appended to the end of the search string. The result of this concatenation operation is assigned to the string parameter passed into the function; the result of this assignment statement, however, has the effect of completely reassigning the value of this parameter. The starting position where the string was found is stored and the `for` loop is exited.

The final statement in the function returns the string parameter passed into the function, which, if no matching strings were found, has the same value as it did when it was passed into the function. However, if a string pattern was found, the replaced string value has been stored in this string. Also returned is the position where the string was found, or, if no string was found, the value zero is returned.

```
function replace_str( find_str, targ_str, str )
define find_str, targ_str, str char(100)
define n, x, y, len_find, len_targ, len_str, found_at smallint
define tmpstr1, tmpstr2 char(100)


let len_targ = length( targ_str clipped )
let len_find = length( find_str clipped )
let len_str  = length( str clipped)

let found_at = 0

for n = 1 to len_str - len_find
```

```
let x = n
   let y = n + len_find - 1

   if find_str clipped = str[ x, y ]  then      -- found it, ⇐
    now replace it

      --
      -- if it's a single char at the first byte
      -- it's a special case
      --
      if x = 1 then
          let str = targ_str clipped, str[ ( y + 1 ), len_str ]
      else
           let str = str[ 1, ( x - 1 ) ],
                     targ_str clipped, str[ ( y + 1 ), len_str ]
      end if

      let found_at = x

      exit for

   end if

end for


--
-- return string and position where substitution started
--
return str clipped, found_at

end function
```

BUBBLE SORT ROUTINE

The `bsort` routine provides an INFORMIX-4GL implementation of the bubble sort. Tapping some of the more obscure features of INFORMIX-4GL string comparison facilities, the bubble sort is surprisingly simple to implement in this language.

The `bsort` routine requires an array of strings to sort, though additional elements could be sorted by simply adding the logic to the comparison section of the code. Since arrays cannot be passed as parameters, they are declared with module scope.

This example shows sample code that calls the `bsort` routine. The array is loaded with values and then the `bsort` routine is called passing the number of elements to sort

as a parameter. Several different types of data are passed to validate the sorting capabilities. The `bsort` routine currently sorts on ASCII collating sequence, though support for integers or other numeric types could be added by changing the data types of the elements of the `sarr` array used in the sort routine.

A bubble sort requires that an array of values be read sequentially. During the read operation, the current element is compared to the next element. If the sort is an ascending sort and the current element is greater than the next element, a swap is required. In a swap the current element is switched with the next element in the sequence. This operation is repeated for each set of elements that meets the criteria for a swap. If at the end of a pass a swap has been performed, then the array must be read again since more swaps may be required. This is continued until a pass of the array is made in which no sorts were required.

The `bsort` routine is passed a parameter which indicates the number of array elements to sort. Some internal flags and counters are declared, as well as a temporary string needed to perform the swap operation. A swapped flag will be maintained to indicate whether or not a swap has been performed in the previous pass of the data.

The swapped flag is set to TRUE at the start of the `while` loop within which the bubble sort will be performed. The `while` loop will continue, meaning the array will be continually scanned, as long as the swapped flag is set to TRUE.

Within the `while` loop, the data value comparisons are performed. The statement

```
if sarr[ n ] > sarr[ n + 1 ] then
```

will perform sensibly in INFORMIX-4GL. Even though the `sarr` array is composed of elements of 20-byte strings, the `if / then` statement will scan all the strings and provide sensible lexical analysis so that the comparison

```
if "Taylor" > "Smith" then
```

will test FALSE and require a swap operation in the `bsort` routine.

The swapping of data elements requires the use of a temporary variable. The current element will be stored in a temporary variable, the next element assigned to the current element, and then the value of the temporary variable assigned to the next element. When a swap occurs, the swapped variable is set to TRUE. The `while` loop will continue as long as a swap occurs. When a loop completes with no swaps, control will exit the loop. Upon completion of the `while` loop, the function returns to the calling function without returning any values.

```
-- module scope array to store sort data
define sarr array[20] of char(10)
```

```
main
define n smallint

   let sarr[9] = "zz"
   let sarr[8] = "ab"
   let sarr[7] = "ac"
   let sarr[6] = "dd"
   let sarr[5] = "ee"
   let sarr[4] = "HH"
   let sarr[3] = "ii"
   let sarr[2] = "jj"
   let sarr[1] = "kk"
   let sarr[10] = "AA"

   display "Before: "

   for n = 1 to 10
       display "n ",n, " value: ",sarr[n]
   end for

   --
   -- sort 10 elements of the array
   --
   call bsort( 10 )

   display "After: "

   for n = 1 to 10
       display "n ",n, " value: ",sarr[n]
   end for


   prompt "Press return to continue " for n

   let sarr[1] = "10"
   let sarr[2] = "09"
   let sarr[3] = "08"
   let sarr[4] = "07"
   let sarr[5] = "06"
   let sarr[6] = "05"
   let sarr[7] = "04"
   let sarr[8] = "03"
   let sarr[9] = "02"
```

```
  display "Before: "

     for n = 1 to 10
         display "n ",n, " value: ",sarr[n]
     end for

     call bsort( 10 )

     display "After: "

     for n = 1 to 10
         display "n ",n, " value: ",sarr[n]
     end for

     prompt "Press return to continue " for n
     let sarr[1] = "Smith"
     let sarr[2] = "Jones"
     let sarr[3] = "Capriati"
     let sarr[4] = "Hari"
     let sarr[5] = "Walker"
     let sarr[6] = "Calder"
     let sarr[7] = "Abbot"
     let sarr[8] = "Jones"
     let sarr[9] = "Taylor"

     display "Before: "

     for n = 1 to 10
         display "n ",n, " value: ",sarr[n]
     end for

     call bsort( 10 )

     display "After "

     for n = 1 to 10
         display "n ",n, " value: ",sarr[n]
     end for

     prompt "Press return to continue " for n

end main
```

```
------------------------------------------------------------

function bsort( numelements )
define numelements smallint
define n, swapped  smallint
define tmp char(10)

let swapped = TRUE
while swapped
  let swapped = FALSE

    for n = 1 to ( numelements - 1 )

       if sarr[ n ] > sarr[ n + 1 ]  then   -- swap them

              let tmp           = sarr[ n ]
              let sarr[ n ]     = sarr[ n + 1 ]
              let sarr[ n + 1 ] = tmp
              let swapped       = TRUE

         end if

       end for

end while

end function
```

The Soundex Routine

The *soundex* routine is a name for a technique of providing a lookup key for a name based on mnemonics instead of the alphabetical spelling of the name. Using the soundex key for lookups provides easy access to the names when the exact spelling of the name is not known and, since it shortens the width of the key for these fields, could speed access to the data.

The `soundex` function takes as its single parameter the string that will be converted to the soundex code. It will ultimately return a string which will be a combination of a single alphabetical character followed by three digits.

The `soundex` function takes the character target string parameter and shifts it to upper case. It also stores the length of the string, since it will be necessary to perform several scans of it. A special case is created for the blank or empty string (which in INFORMIX-4GL would be the NULL string). In this case, the function will return a blank succeeded by three zeros.

The first scan starts at the second byte of the string, since the first byte will be retained for the first byte of the soundex key. This scan searches for certain characters which will later be thrown out. If a vowel character is found in the string, it is flagged for deletion by placing a character '9' in that position. The next loop searches for specific sets of consonants that are mapped to digits. The consonants in the target string are replaced by these digits.

The next `for` loop is used to strip duplicates out of the target string. The string is scanned with a `for` loop that moves two bytes per loop iteration. For each iteration of the loop, the current byte is compared to the succeeding byte in the string; if they match, the succeeding byte is flagged for deletion by assigning a character '9' to that position.

Now the bytes from the target string can be copied to the return string. Characters flagged for deletion can be eliminated by simply not copying bytes with the character '9' in that position.

The final loop is used to pad the string. Any blank bytes are to be padded with the digit zero. After completing the final loop, a substring of the return string from bytes one through four is returned to the calling function.

```
main
define str array[15] of char(20)
define ret_val char(4)
define x,n smallint

let str[1] = "Smith"
let str[2] = "Smythe"
let str[3] = "Smethe"
let str[4] = "Jones"
let str[5] = "James"
let str[6] = "jane"
let str[7] = "Taylor"
let str[8] = "Tyler"
let str[9] = "Tipper"
let str[10] = "Tanya"
let str[11] = "Lleperheimer"
let str[12] = "Leperheimer"
let str[13] = "Flemington"
let str[14] = "Flemingtown"
let str[15] = "Phlemington"

for n = 1 to 15
   let ret_val = soundex( str[n] )
   display "Returned : ", ret_val, " for ", str[n],  " - ", n
end for
```

```
prompt "Press return to continue " for n

end main

--------------------------------------------------------------------

function soundex( str )
define str char(64)
define ret_val char( 64 )
define n,x, len smallint
define s char(1)

let str = upshift( str )
let len = length( str )

if len = 0 then
   return " 000"
end if

for n = 2 to len
   if str[n] matches "[AEHIOUWY]"  then
      let str[n] = "9"        # flag this character for ⇐
    later deletion
   end if
end for

for n = 2 to len

   if str[n] matches "[BFPV]"  then
      let str[n] = "1"
      continue for
   end if

   if str[n] matches "[CGJKQSXZ]" then
      let str[n] = "2"
      continue for
   end if

   if str[n] matches "[DT]" then
      let str[n] = "3"
      continue for
   end if
```

```
   if str[n] = "L" then
      let str[n] = "4"
      continue for
   end if

   if str[n] matches "[MN]" then
      let str[n] = "5"
      continue for
   end if
   if str[n] = "R" then
      let str[n] = "6"
      continue for
   end if
end for

-- strip out duplicates
for n = 2 to len step 2
   if str[n] = str[n + 1] and
      str[n] != "9"        then  # it's a dupe
      let str[ n + 1 ] = "9"     # flag for deletion
   end if
end for

-- now copy to target, eliminating those chars flagged ⇐
    for deletion
let x = 1
for n = 1 to len
    if str[n] != "9" and str[n] != " "   then    # ok to copy
       let ret_val[x] = str[n]
       let x = x + 1
    end if
end for

--
-- pad the string, if necessary
-- above loop left us at one byte past that which we copied
-- need to copy to that byte
--
if x <= 4  then          -- if we are NOT looking at the 4th byte,
                         -- then we need to pad the return string
  for n = x to 4         -- pad the string
      let ret_val[ n ] = "0"
```

```
  end for
end if

return ret_val[ 1, 4 ]

end function
```

Chapter 15

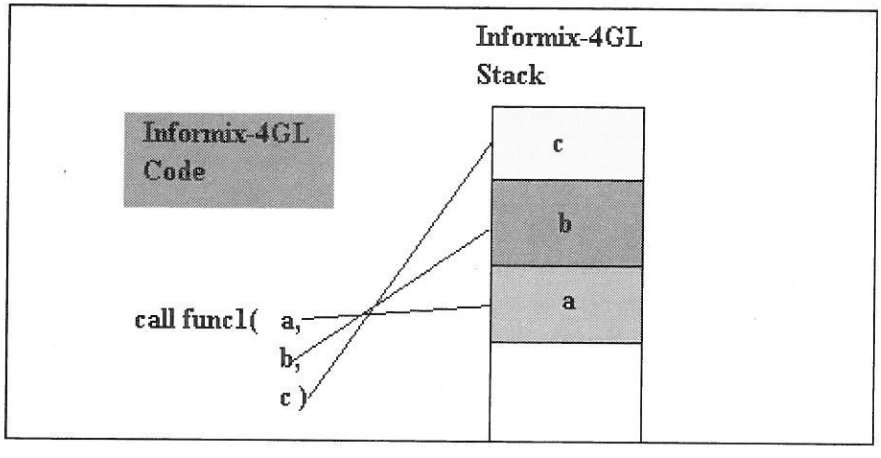
INFORMIX-4GL C-Language Wrapper Functions

While it is loaded with a number of very useful features, the INFORMIX-4GL language does lack the extensive operating system interface of lower-level languages such as the C programming language. This was a conscious deletion; the language was designed to aid report writing, write data entry screens and provide for interaction with a database. It was not intended to write device drivers, process flat files and develop operating systems, as was the C language. Nevertheless, access to file input/output facilities is often a necessary feature in applications that must communicate with other computer systems and databases. This chapter presents several C-language functions that have been modified to allow INFORMIX-4GL applications to access these functions.

THE INFORMIX-4GL STACK

INFORMIX-4GL provides an easy access to the C programming language through a set of stack functions. Before a C-language function can be called, the parameters being passed to the function must be placed or *pushed* on the INFORMIX-4GL stack. This happens automatically when the function is called from the INFORMIX-4GL program. The function call can be either explicit, with the `call` statement, or implicit when the function is used in an expression.

Figure 15-1 The INFORMIX-4GL Stack



In the C-language function being called, the arguments must be retrieved or *popped* from the stack and stored in C-language variables. This is done with the pop and return functions provided by INFORMIX-4GL. These functions are shown in Table 15-1.

Table 15-1 INFORMIX-4GL Pop Stack Manipulation Functions

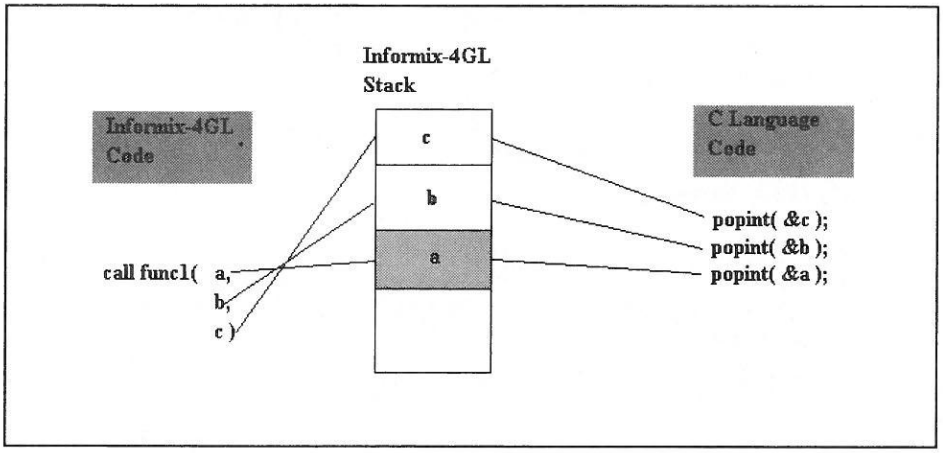
Function Name	Description	C-Language Parameter	Calling Format
`popint`	Pop integer value off stack. Passed address of integer value.	`int i;`	`popint(&i);`
`popshort`	Pop short integer value off stack. Passed address of short integer value.	`short i`	`popshort(&i) ;`
`poplong`	Pop long integer value off stack. Passed address of long integer value.	`long i;`	`poplong(&i);`
`popflo`	Pop float value off stack. Passed address of float value.	`float f;`	`popflo(&f);`
`popdub`	Pop double float value off stack. Passed address of double float value.	`double f;`	`popdub(&f);`
`popdtime`	Pop datetime value off stack. Passed address of datetime structure and an integer qualifier value.	`dtime_t dt; int q;`	`popdtime(&dt , q);`
`popinv`	Pop interval value off stack. Passed address of interval structure and an integer qualifier.	`intrvl_t intrvl; int q;`	`popinv(&intr vl,q);`
`popquote`	Pop character string off stack. Passed address of character string and an integer length value.	`char *str; int n;`	`popquote( s, n );`
`popdec`	Pop decimal value off stack. Passed address of decimal structure.	`dec_t dm;`	`popdec( &dt );`

Function parameters are popped in a right-to-left fashion relative to the order in which they were passed in the function call. If a function was called with

```
call this_func( a, b, c )
```

the parameters would be popped would be c, then b and then a, as shown in Figure 15-2.

Figure 15-2 INFORMIX-4GL Stack



There are no restrictions on the code that can be used in the C-language function that has been called. However, on completion of the function, any parameters to be returned to the INFORMIX-4GL calling function must be placed on the INFORMIX-4GL stack using the *push* or return functions shown in Table 15-2.

Table 15-2 INFORMIX-4GL Return Stack Manipulation Functions

Function Name	Description	C-language Parameters	Calling Format
retint	Returns an integer value. Passed an integer variable.	int i;	retint(i);
retshort	Returns a short integer value. Passed a short integer variable.	short i;	retshort(i);
retlong	Returns a long integer value. Passed a long integer variable.	long i;	retlong(i);

continued on next page

Table 15-2 continued

Function Name	Description	C-language Parameters	Calling Format
`retflo`	Returns a floating point value. Passed an address of a floating point variable.	`float f;`	`retflo( &f );`
`retdub`	Returns a double float value. Passed the address of a double variable.	`double f;`	`retdub( &f );`
`retquote`	Returns a NULL-terminated string value. Passed an address of a NULL-terminated string variable.	`char *s;`	`retquote( s );`
`retdec`	Returns a decimal value. Passed an address of a decimal structure.	`dec_t d;`	`retdec( &d );`
`retdtime`	Returns a datetime value. Passed the address of a datetime structure.	`dtime_t dt;`	`retdtime(&dt);`
`retinv`	Returns an interval value. Passed the address of an interval structure.	`intrvl_t it;`	`retinv( &it );`

Return values must be placed onto the stack in a left-to-right fashion, the same order in which they appear in the `returning` statement of the INFORMIX-4GL program that called the function.

In addition to calling the appropriate INFORMIX-4GL functions to place the return value on the stack, the C-language `return` statement must be used with an integer parameter which indicates to the calling function the number of parameters that have been placed on the stack and are now being returned. The following library code examples demonstrate this interface.

INFORMIX-4GL C Library Access Functions

The functions listed in Table 15-3 provide an INFORMIX-4GL language access to some common C library functions through INFORMIX-4GL function calls. These are

effectively *wrapper* library functions. Their names and usage are specified below. (Note: `fgl_xyz` names are reserved by INFORMIX-4GL for future use.)

Table 15-3 C-language Wrapper Functions

Function Name and Parameters	Description
`fgl_open( fname, open_mode)`	Opens an operating system file. Returns an integer file number.
`fgl_fgets( fnum, number_bytes )`	Reads a character string from a file up to the number of bytes limit passed. If the number of bytes is 0, will read up to the maximum buffer size or the newline character. Returns the number of bytes read.
`fgl_fputs( buff, bufflen, fnum)`	Writes the character string buffer passed to the file up to the length of the buffer. Returns the number of bytes successfully written.
`fgl_close( fnum )`	Closes the file number specified. Returns 0 if successful, or negative 1 if an error occurred.
`fgl_unlink( fname )`	Unlinks or removes the file name passed. Returns a success code.
`fgl_rand()`	Returns an integer random number. Uses the operating system time as a seed.
`fgl_getpid()`	Returns the integer process id of the current process.
`fgl_rename( from_name, to_name )`	Renames the file from the character string name passed to the new character string name passed. Returns the success code of the rename operation.

THE C LIBRARY WRAPPER FUNCTION CODE

The first section of code in the library is used for the C-language `include` statements. These statements are required to call several of the C-language library functions used in the function library. These `include` statements could change depending on the platform where the code is being compiled. In addition to the `include` statements, a BUFFSIZE parameter is defined. This is the maximum buffer size used in the file operations.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#include <stdio.h>
#include <stdlib.h>

#define BUFFSIZE 500
```

The fgl_open Function

The fgl_open function is passed the name of a file to open and the mode with which to open the file. Both parameters are passed as character strings. The function returns an integer file number which will be used to reference the file in future function calls. It is called as follows.

```
define outfile integer

call fgl_open( "appl1.out", "w" )
      returning outfile
```

Subsequent operations on the file will pass the file number (contained in the integer variable outfile) to the functions used to perform file operations.

The fgl_open C function is declared to receive a single C-language function parameter — the number of parameters passed into the function. This declaration is required for all C-language functions that are called from INFORMIX-4GL applications. The C-language function can use this for two purposes. One purpose is to validate that the expected number of parameters has been passed. The second is to allow a variable number arguments to be received by the function. Such functions are frequently written so that if parameters have not been passed, default values will be substituted. In this function, the number of parameters value is simply used to validate that the correct number of parameters has been passed.

If the correct number of parameters has not been passed, the function returns a negative 1, indicating to the calling function that an error has occurred. Since this code is written in the C language, there is no practical access to INFORMIX-4GL error printing facilities. Also, since INFORMIX-4GL often places the terminal in raw mode, C-language terminal output facilities may not perform as expected. The most practical and most common method of error handling when calling C functions from within INFORMIX-4GL is to have the C function quietly return an error code to the calling INFORMIX-4GL function and allow the calling function to signal the error to the user.

After the number of parameters has been evaluated in the fgl_open function, the parameters are popped from the stack. Since they are both string variables, the popquote function is used to retrieve both parameters.

The popquote function pads string values to their defined length. In this case, we have specified fifty characters for the length of the file name, so the file name string will be blank—padded to 50 characters. If the string was a 10-byte file name and was

retrieved from the stack with a call to popquote that specified a maximum of 50 characters to retrieve, then the string would be padded with thirty blanks, from the eleventh byte to the fiftieth byte.

Since most C library functions do not expect strings to be blank padded, the `clip_it` function (shown later) is called to remove the blank padding from the file name variable.

The string variable for mode was popped from the stack with a length specification of 2 bytes. The minimum string length in INFORMIX-4GL is two, with 1 byte used to store a single character, and the second byte used for the NULL terminator.

Next, the first byte of the mode variable is evaluated. This indicates the mode in which the file will be opened. The file open mode is mapped to several specific modes: "r","w" or "a". These modes are explained below.

If an "r" has been passed as a parameter, the file is opened in *read-only* mode. If the file is not present when the open operation is executed, then the C library `open` function call will fail and a negative 1 will be returned to the calling function. Otherwise the file will be opened in read-only mode.

If a 'w' has been passed for the file open mode, then the file will be opened in a *write-only* mode. If the file is not present, it will be created with a read/write file mode. The number returned from the C library `open` function will be returned to the calling function.

If an 'a' has been passed for the file open mode, then the file will be opened in an *append* mode. If the file is not present, then it will be created and given a read/write file mode.

If the mode passed was not an "r", "w" or "a", then control will fall through the `switch` statement and return the initial integer value of the file number host variable. This will lead to a 0 being returned to the calling function. The calling function must be aware that the value of 0 from the `fgl_open` function call indicates an end of file or an error condition.

```
/* fgl_open( char fname, char mode ) returns fnum  */
fgl_open( num_params )
int num_params;
{
int fnum;
char mode[2];
char fname[50];

if ( num_params != 2 ) {
   retint ( -1 );
   return( 1 );
}
```

```
popquote( mode, 2 );
popquote( fname, 50 );

clip_it( fname, 50 );  /* get rid of blank padding */

fnum = 0;
switch( mode[0] ) {
   case 'r': fnum = open( fname, O_RDONLY );
             break;
   case 'w': fnum = open( fname, O_WRONLY | O_CREAT, ⇐
    S_IREAD | S_IWRITE );
             break;
   case 'a': fnum = open( fname, O_APPEND | O_CREAT, ⇐
    S_IREAD | S_IWRITE );
              break;
}

retint( fnum );
return( 1 );

}
```

The `fgl_fputs` Function

The `fgl_fputs` function receives a string buffer, the length of the buffer and a file number as parameters. The function takes the buffer and outputs it to the file that corresponds to the file number passed. It will output the buffer up to the length specified unless a 0 was passed as the buffer length. If a 0 buffer length was passed, it will determine the length of the buffer without trailing blanks (the *clipped* length) and output the buffer up to that length.

The function first evaluates the number of parameters passed. If the number of parameters is not 3, then it returns a negative 1 indicating that an error has occurred. If the correct number of parameters has been passed, the function continues execution. It first pops the file number parameter and then the buffer length parameter.

How the function pops the final parameter from the stack is dependent on the value of the buffer length parameter. If the buffer length parameter is 0, it will pop the buffer from the stack up to the defined maximum length of the buffer as defined in the BUFFSIZE constant. It will then try to trim the buffer of trailing spaces. If the buffer length is non-0, it pops the buffer off the stack using the buffer length as a parameter. This logic optimizes the operation of popping the parameter from the stack when the buffer length is passed, since only the necessary number of bytes will be popped from the stack.

Once the correct buffer length is determined, the buffer is NULL-terminated at that point less 1 byte to coincide with C-language arrays which start at element 0 instead of element 1 as with INFORMIX-4GL arrays. The C-language `write` function is then called to output the buffer to the file corresponding to the file number passed. The buffer will be output up to the buffer length specified.

The number of bytes successfully written is returned by the `write` function. If this number does not match the number of bytes in the buffer, a negative 1 will be returned. Otherwise, the number of bytes written (as stored in the `out` variable) is returned to the calling function.

```
fgl_fputs( num_params )
int num_params;
{
char buff[BUFFSIZE];
int fnum,n, out, buflen;
char *c;

if ( num_params != 3 ) {
   retint( -1 );
   return( 1 );
}

/*
 * get the values from the stack
 */
popint( &fnum );
popint( &buflen );

if ( buflen == 0 )    /* user didn't specify buffer length */
   popquote( buff, BUFFSIZE );
else                  /* user specified length. Use it in popquote */
   popquote( buff, buflen );

if ( buflen == 0 ) {

  /*
   * determine clipped size of string
   */
   c = &buff[BUFFSIZE-1];
   n = BUFFSIZE;

   while ( ( ( *c > '~' ) || ( *c <= ' ' ) ) && ( n > 0 ) ) {
         c--;
```

```
            n--;
    }
    buflen = n;
}
else {                          /* user gave us the length */
    c = &buff[ buflen - 1];
    n = buflen;
}

*c = '\n';

out = write( fnum, buff, buflen );
if ( out != buflen ) {
    retint( -1 );
    return( 1 );
}
else {
    retint( out );
    return( 1 );
}

}
```

The `fgl_fgets` Function

The `fgl_fgets` function retrieves buffers from the file number passed and returns them to the calling function. It is passed the number of the file as returned by the `fgl_open` function, and the number of bytes to retrieve from the file. It retrieves the buffer from the current file position up to the number of bytes requested or until a newline character is found and then returns the buffer.

The function first validates that the correct number of parameters has been passed into the function. If the correct number of parameters has been passed, the function pops the number of bytes to read and the number of the file to read from the stack. It then executes the C library `read` function to read the buffer from the file specified using the parameters just retrieved from the stack.

The INFORMIX-4GL function that has called this function expects a NULL-terminated string as a return parameter. The C library `fgets` function returns NULL-terminated strings with newline characters. This implementation of an `fgl_fgets` function will map the newline character to a NULL terminator to make the return value more appropriate for an INFORMIX-4GL application.

This mapping is accomplished with the `while` loop, which begins at the end of the string buffer and scans towards the start of the buffer looking for a newline character. It starts at the end of the string since it is most probable that the newline character is at or

near this address, thus reducing the number of loop iterations required to perform the mapping.

Once the NULL terminator has been placed in the return buffer, code is executed to determine the number of bytes that are being returned. If the address where the NULL terminator is located is different from the buffer length passed into the function, that address is assigned the new buffer length. Otherwise, a NULL terminator is placed at the end of the buffer string (to allow for conditions where a string was retrieved with no newline character), and the buffer is returned with a `retquote` function.

At this point, it is necessary to synchronize the file pointer for the current file with the buffer that is being returned. The file pointer points to the location of the end of the buffer specified with the buffer length when the parameters were passed in. However, since this function will attempt to return records delimited by newline characters, the buffer returned may be shorter than the bytes requested, and the file pointer may not point to the start of the next record. The statement that calls `lseek` will set the file pointer to the correct position.

```
fgl_fgets( num_params )
int num_params;
{
static char buff[BUFFSIZE];
int fnum, num_bytes, bytes_ret, bytes_read, n;
static char *c;

/*
 * validate that the correct number of parameters has been passed
 */
if ( num_params != 2 ) {
   retquote( "" );
   return( 1 );
}


 /*
 * retrieve the parameters
 */
popint( &num_bytes );
popint( &fnum );

/*
 * read the data
 */
bytes_read = read( fnum, buff, (unsigned )num_bytes );
```

```
/*
 * if nothing read, then return
 */
if ( ( bytes_read == -1 ) || ( bytes_read == 0 ) ) {
   retquote( "", 1 );
   return( 1 );
}

/*
 * get address for end of string
 */
c = &buff[0] + bytes_read;

/*
 * scan backwards
 * make the newline a NULL
 */
while ( ( *c != '\n' ) && ( c != &buff[0] ) )
      c--;

/*
 * if we didn't decrement all the way to the start of the string
 * then set this address value to NULL
 */
if ( c != &buff[0] ) {
   *c = '\0';
   bytes_ret = c - &buff[0];
}
else   {                          /* then just null terminate */
   buff[ bytes_read ] = '\0';  /* allow for NO newline in string
*/
   bytes_ret = bytes_read;
}


if ( bytes_ret == 0 )  {
      retquote( "", 1 );
      return( 1 );
   }

/*
 * set file pointer to the start of the next line
 */
```

```
 if ( ( bytes_read - bytes_ret ) > 0 )
    n = lseek( fnum, (-1 * ( bytes_read - bytes_ret - 1)),⇐
      SEEK_CUR );

retquote( buff );
return(1);

}
```

The `fgl_close` Function

The `fgl_close` function first evaluates the number of parameters passed. If the number of parameters passed is not 1, it returns an integer negative 1 to indicate that an error has occurred. If a single parameter was passed, the function pops the file number parameter from the stack. It then executes a C library `close` function using the file number and returns the integer value returned by the `close` function.

```
fgl_close( num_params )
int num_params;
{
int fnum;

if ( num_params != 1 ) {
    retint( -1 );
    return( 1 );
}

popint( &fnum );

retint( close( fnum ) );
return( 1 );

}
```

The `fgl_unlink` Function

The `fgl_unlink` function receives a single parameter: the name of the file to remove or unlink. It mirrors the functionality of the C library `unlink` function.

The `fgl_unlink` function first evaluates the number of parameters passed into the function. If the number of parameters is not 1, it signals an error message by returning a negative 1 to the calling function. If the correct number of parameters has been passed, the file name parameter is popped from the stack.

The `popquote` function will retrieve a string blank-padded to the string length passed to the function. Since the C library function would not evaluate the file name

correctly in a blank-padded string, the string is trimmed of trailing blanks using the `clip_it` function. The C library `unlink` function is then called to remove the file (*unlink* its entry from the directory linked list). The value returned by the `unlink` function is then returned to the calling function.

```
fgl_unlink( num_params )
int num_params;
{
char fname[50];
char *c;
int n;

/*
 * validate the number of parameters
 */
if ( num_params != 1 ) {
   retint( -1 );
   return( 1 );
}

/*
 * retrieve the file name
 */
popquote( fname, 50 );

/*
 * clip the string
 */

clip_it( fname );
retint( unlink( fname ) );

return( 1 );

}
```

The `fgl_rename` Function

The `fgl_rename` function takes two file name parameters: the old file name and the new file name. It will rename the old file to the new file name.

The `fgl_rename` function takes two arguments: the current name of a file and a new name for the file. The function first evaluates the number of parameters passed; if the number is not 2, it returns a negative 1, indicating that an error has occurred. If the correct

number of parameters has been passed, the function will use the `popquote` function to retrieve the two file name parameters from the stack. The `clip_it` function is then used to trim these strings of trailing blanks. The strings are then used as parameters to the C library `rename` function. The return value of the `rename` function is returned to the INFORMIX-4GL calling function.

```
fgl_rename( num_params )
int num_params;
{
char from_name[50], to_name[50];
char *c;
int n;

if ( num_params != 2 ) {
   retint( -1 );
   return( 1 );
}

popquote( to_name, 50 );
popquote( from_name, 50 );

clip_it(from_name, 50);  /* get rid of the blank padding */
clip_it( to_name, 50 );

retint( rename( from_name, to_name ) );
return( 1 );

}
```

The `fgl_rand` Function

The `fgl_rand` function returns a random number generated using the C library `rand` function. It receives no parameters. If any parameters have been passed into the function, the function returns a negative 1.

On the first pass into the function, it will seed the random number generator. It calls the C-library `time` function to return the current time and then uses the time as a seed to the C library `srand` function. (This will help yield a more random pattern of numbers on each call.) The integer value returned from the C-library `rand` function is then returned to the INFORMIX-4GL calling function.

```
fgl_rand( num_params )
int num_params;
{
```

```
int ct, rn;
static short first_pass = 1;

if ( num_params != 0 ) {
   retint( -1 );
   return( 1 );
}

if ( first_pass ) {
/*
 * use the current time as a seed
 */
time( &ct );
srand( ct );
first_pass = 0;
}

/*
 * return the random number
 *
retint( rand() );
return( 1 );

}
```

The `fgl_getpid` Function

The `fgl_getpid` function simply returns the process id of the current process. It receives no parameters. It will evaluate the number of parameters passed and, if any have been received, it will return a negative 1 to indicate an error. If no parameters have been received, it will return the value of the C library `getpid` function.

```
fgl_getpid( num_params )
int num_params;
{

if ( num_params != 0 ) {
   retint( -1 );
   return( 1 );
}

retint ( getpid() );
return( 1 );

}
```

The `clip_it` Function

The `clip_it` function is used internally by the C library functions just detailed. It takes a character pointer and an integer number of bytes as parameters. The number of bytes represents the padded length of the string. The object of this function is to scan from the end of the string backwards until the first nonblank character is found. When the first nonblank character is found, the pointer will be incremented and a NULL character placed at that position in the string.

In order to accomplish this, the function first increments the character pointer passed into the function by the number of bytes in the character string, thus establishing a pointer to the end of the string. Starting from this pointer, the `while` loop will scan backwards, decrementing the pointer and examining each character during the process. If the character falls outside of therange of *blank* characters, or the loop has scanned the entire string, the `while` loop will terminate. Since the `while` loop will most likely terminate when pointing at a nonblank character, it is incremented to move past this nonblank character. A NULL terminator is then placed at this point, and the function returns to the calling function.

```
clip_it( str, bytes )
char *str;
int bytes;
{
char *c;
register int n;

c = str + bytes - 1;
n = bytes;
while ( ( ( *c > '~' ) || ( *c <= ' ' ) ) && ( n > 0 ) ) {
      c--;
      n--;
}

*++c = '\0';

}
```

SUMMARY

With very little effort, the INFORMIX-4GL language can be expanded to include the power of the C programming language and the rich set of library functions available within that language. The parameters passed into these functions must sometimes be converted to a format that the C library functions will understand, such as the padded strings that must be clipped of trailing blanks. Though not shown here, working with datetime and intervals also requires conversion. However, once converted, these data values can be used just as any other C-language variable.

Index

—A—

after group of, 85
all function
  example, 107
applications
  Calculator, 191
  calendar, 209
Arr_curr, 151
arrays
  parallel, 193
arrow keys, 167

—B—

Back-end, 3
Block, 2
Block-oriented statement, 6
Boolean expressions
  nulls, 14
Bsort
  impemenatoin, 160
bsort2b, 168

—C—

c language access, 335
C language wrapper functions, 335
Calculator, 191
Calendar, 209
Call-by-value, 111
center function
  example, 107
character processing, 191
Clip_it
  explained, 351
constants
  Lack of, 20
  substitute for, 20
Construct, 241
Conversion, 18
current field
  Highlighting, 64
Cursors, 35, 37, 136
  Closing, 45
  identifiers, 45
  insert, example, 44
  locking, 42
  Scroll, 38
  scroll, example, 38
  scroll, problems, 40
  scroll, stale data, 40
  Types, 37
  Update, 42
  update, performance advantage, 42

—D—

data
  Initialization, 20
Data conversion, 3, 16, 18
  Character, 18
  numeric, 19
data types, 11
  Arrays, 12
  Character, 12
  Complex, 12
  Date, 10
  Datetime, 10
  Decimal, 8
  Float, 10
  Integer, 8
  Interval, 11
  Money, 9

Records, 13
Smallfloat, 8
Smallint, 7
Data typing, 3
Database binding, 13
define, 27
Database bound, 297
Database Interaction, 3
Date, 10
Datetime, 10
Decimal, 8
Decision operations
optimizing, 300
default
report, 74
define
database binding, 27
delete key
Turning off, 66
design
modular, 192
display
by name, 60
unformatted output, 59
display array, 149
Display statement, 58
dynamic
reports, 100
reports, control table, 103
Dynamic reports, 93

—E—

end of array
enforcing, 161
error, 268
functions, 271
logging, 268
Error handling, 265
Errors, 265
date, 265
decimal, 265
err_get, 271
err_print, 271
err_quit, 271
error report, 266
errorlog, 271
expression, 270
forms, 265, 270
generalized error handlers, 272
production environment, 266
SQL, 265
sqlca record, 276
startlog, 268, 272
syntax, 265
trapping, 266
whenever error, 266
expressions
Boolean, 22
Boolean, evaluation of, 22

—F—

Fgl_close, 339
explained, 347
Fgl_fgets, 339
explained, 344
Fgl_fputs, 339
explained, 342
Fgl_getpid, 339
explained, 350
Fgl_open, 339
explained, 340
Fgl_rand
explained, 349
Fgl_rename, 339
explained, 348
Fgl_unlink, 339
explained, 347
Float, 10
FOREACH, 35
Form4gl, 49
FORMONLY, 50, 218
Forms errors, 270
Front-end, 3, 298
Function, 2

library, 281
multiple parameters, 111
functions, 111
all, 320
bsort, 326
call-by-value, 111
center, 319
err_handler, 293
err_msg, 274
errorlog, 268
expressions, 113
find_str, 322
format, 111
gen_tab_input, 219
Generic, 137
Implementation, 112
infield, 276
is_number, 321
is_print_ascii, 321
Library, 317
library, center, example, 107
nesting, 211
Parameters, 111
prompt_msg, 321
Recursion, 119
replace_str, 324
run_calc, 194
show_cal, 209
showhelp, 275
soundex, 330
stack, 112
startlog, 268
Variable number of parameters, 116
variable parameters, 137, 140
variables, 114

—G—

Gen_tab application, 217
generalized error handlers, 272
Generalized table browser, 217
Generic functions, 137
Issues, 139
Global array of records, 158

—H—

help file, 276
Help messages, 265, 274
Highlighting, 64

—I—

infield, 276
Informix-4GL development environment, 311
Informix-4GL stack, 335
Informix-Perform, 5
Initialization, 20
INPUT, 62
clauses, 63
Input array, 149
sidewarys, 170
input field
moving past, 63
insert key
Turning off, 66
Integer, 8
integration
code, 192
Interprocess communication, 37, 298
interrupt key
trapping, 68
interrupt signal
Trapping, 68
Interval types, 11
intervals
Day/Hour, 12
Year/Month, 11
IPC, 37, 298

—L—

Library, 281, 317
organization, 311

loadrec(), 179
Local scope, 114
Loop operations, 301

—M—

make
  suffix, 312
  suffix rules, 313
Make utility, 312
  creating objects, 314
  Source code control, 315
  Supporting p-code and compiled versions, 314
  use of, 312
  using with Informix-4GL, 312
Mask, 26
memory control, 3
mkmessage, 275
Modular, 192, 281
Modular application development, 6
Modularity
  SQL, 47
Module scope, 48, 114
Money, 9
moving back a page, 255

—N—

Non-procedural, 1
Null data types, 14

—O—

operator
  Concatenation, 23
  modulo, 26
Operators, 2
  Arithmetic, 19
  evaluation, 19
  grouping, 19
  precision and scale, 20
Optimization
  prepare statement, 31
  Reducing string operations, 304
optimizing
  Eliminating parameter passing, 307
  Eliminating statements and expressions, 303
  function calls, 302
  loop operations, 301
  parameter passing, 306
  repetitive code, eliminating, 300
  SQL, 308

—P—

page breaks, 84
parameters
  functions, 111
  reports, 88
parsing
  delimited string, 116
parts explosion problem, 132
performance, 297
  insert cursors, 43
pick list
Popdec, 336
Popdtime, 336
Popdub, 336
Popflo, 336
Popint, 336
Popinv, 336
Poplong, 336
Popquote, 336
Popshort, 336
Prepare, 265
PRINT, 82
  using clause, 82
Procedural, 1
Program compilation
  automating, 312
programming
  Screen form, 49
prompt

for characte, 191
push, 335

—R—

Recursion, 119
cursors problem, 136
parts explosion, 132
Redisplayed, 166
Repetitive code
identifying, 300
Report Interaction Statements, 71
reports, 71
AFTER GROUP OF, 79, 85
aggregate functions, 79
Batch processing, 80
BEFORE GROUP OF, 79
Column positions, 97
construct statement driven, 93
declaring, 73
default, 74
dynamic, 93, 100
dynamic column positions, 78
dynamic parameters, 88
dynamic reports, 78, 86
examples, 100
first page header, 77
format section, 76, 79
format, First page header, 77
initialization in first page header, 77
issues, 88
margin control, 96
modularity, 86, 90
output section, 75
Output section clauses, 75
output, order by, 76
page breaks, 84
PRINT STATEMENT, 82
print statement, using clause, 82
queries stored in tables, 100
report control table, 103
restrictions, 71
statements, 72
Synchronous, 92
text fields, 83
top of page, 84
Retdec, 338
Retdtime, 338
Retdub, 338
Retflo, 338
Retint, 337
Retinv, 338
Retlong, 337
Retquote, 338
Retshort, 337
Rowid access, 218

—S—

Sample development environment, 311
save_rec, 179
scope
Module, 48
Screen form, 5
attributes, 50
database, 51
FORMONLY, 50
identifiers, 52
instructions, 50
Manipulating, 52
Modularity, 52
multiple windows, 53
Screen form programming, 49
Screen forms, 49
Database binding, 50
Screen size
managing, 49
Set_count, 151
showhelp, 275
sideways scrolling, 170
Smallfloat, 8
Smallint, 7
Soundex, 330
Source code module, 4

SQL, 25
  administrative tasks, 25
  Cursors, 35
  database binding, 27
  declare, 32
  DYNAMIC SQL, 29
  Modularity, 47
  Optimization, 31
  parsing, 31
  PREPARE STATEMENT, 29
  statements not supported, 25
  type conversion, 26
sql code values, 277
SQL processing, 31
sqlca record, 276, 277
Sqlexec, 4
Sqlturbo, 4
Stack, 112, 126
  decision operations, 300
  order of parameters, 338
  parameters, 338
  pop, 336
  popped, 336
  push, 335
stack operations explained, 335
startlog
  logfile, 269
Statements, 2
  Display, 58
  for, 21
  FOREACH, 35
  INPUT, 62
  Looping, 21
Suffix, 312
suffix rules
  changing, 313
Synchronous
  reports, 92
Systables, 219
System catalog tables, 219
system catalogs, 25

—T—

Tabid, 219
Table browser, 217
  display_form, 227
  Forward page, 254
  GEN_ADD, 232
  gen_cols table, 227, 230
  GEN_DELETE, 238
  gen_disp, 248
  gen_fetch, 245
  gen_input, 259
  gen_open, 244
  gen_query, 240
  gen_tab_input, 219, 222
  GEN_UPDATE, 235
  globals, 221
  init_flds_arr, 249
  init_mstr_flds, 249
  ld_flds_lbls, 253
  ld_flds_val, 250
  LOAD_FIELDS, 230
  main, 218
  Menu options, 224
  move_flds_page, 254
  scr_page, 258
  screen form, 229
  validate_field, 244
Template, 281
  add option, 283
  add_rec function, 292
  curs_move function, 292
  delete, 283
  delete_rec function, 291
  display_rec function, 288
  err_handler function, 293
  input_rec function, 289
  Next option, 284
  open_curs function, 287
  open_form function, 292
  other applications, 294
  Previous option, 285
  qbe function, 287

query option, 282
upd_rec function, 291
update, 283
termcap, 66
Redefining function keys, 66
termpca
Adding additional function keys, 67
text fields, 83
towers
data structures, 132
Towers of Hanoi, 120
Type conversion
advantage, 217

—U—

using clause
format characters, 16

—V—

validate function, 146
Variable
name scope, 111
Variables
in SQL statements, 35
local scope, 114
module scope, 114
Views, 45

—W—

whenever any error
whenever error
continue, 267
scope, 266
Whenever error continue, 267
window
current, 56
current, example, 56
windows
current, 53
layered, 55
multiple, 53

—Z—

Zoom, 154, 281
zoom function, 149
zoom1 function, 149
Zoom2, 154

DATABASE TECHNOLOGY
Languages

INFORMIX Press

DISKETTE INCLUDED.
DO NOT DEMAGNETIZE

ADVANCED INFORMIX-4GL PROGRAMMING

ART TAYLOR

ADVANCED INFORMIX-4GL PROGRAMMING demonstrates how to use INFORMIX-4GL as a sophisticated programming tool. Taking advantage of many of the advanced features of the language, Taylor shows how to use INFORMIX-4GL to solve complex programming problems previously relegated to "Third Generation Languages" such as C, BASIC or Pascal. Included are INFORMIX-4GL solutions for "The Towers of Hanoi" problem and "The Parts Explosion" problem.

ADVANCED INFORMIX-4GL PROGRAMMING includes numerous examples of complex 4GL code for real-world, general purpose applications such as a calendar, a table browser, and a desktop calculator. Most of the code examples are complete, working applications or functions, and they can be found on the diskette included with the book.

In addition, the book presents a useful toolkit of library functions to perform operations; among them are pick lists, validations, error messages, string searching and replacing, in-memory sorting and mnemonic character string mapping via a 'soundex' type algorithm.

The INFORMIX-4GL interface to the C language is also disc ...wrapper' functions is presented to provide INFORMIX-4GL access to C language libra... unlink for operating system file I/O.

Taylor covers INFORMIX-4GL report writing facilities in det... ent to provide for dynamic report selection criteria is presented. A st... And an example of a generic report driver, which allows each re... queries, is also shown.

Throughout the book, Taylor stresses the benefits of using INFORMIX-4GL's modular, ... e to create structured INFORMIX-4GL programs. These benefits include reduced development time, easier code maintenance, and reduced programmer training time. All of the examples are written in a structured manner.

Art Taylor is a consultant for Informix Software, Inc. He has over 8 years of experience working with Informix products in the UNIX environment. He has worked with INFORMIX-4GL since the product's inception.

6th
25141804 | PQ ID: PQ56964206
1048-5-310-27352-4296164131 | 0123013184
Advanced Informix-4GL Pro
20150306 | 4 | AMZUS-158 | 1 out of 1 items

Other Informix Press titles available from PTR Prentice Hall/Informix Press:

- INFORMIX-4GL by Example, from Informix Software, Inc. ISBN 0-13-100355-0
- Informix Guide to SQL: Tutorial, from Informix Software, Inc. ISBN 0-13-100371-2
- Informix Guide to SQL: Reference, from Informix Software, Inc. ISBN 0-13-100363-1
- Programming Informix SQL/4GL: A STEP- BY- STEP Approach, by Cathy Kipp ISBN 0-13-149394-9
- The Informix Database Administrator's Survival Guide, by Joe Lumbley ISBN 0-13-124314-4
- INFORMIX-OnLine Performance Tuning, by Elizabeth Suto ISBN 0-13-124322-5

PRENTICE HALL
Englewood Cliffs, NJ 07632

ISBN 0-13-301318-9
90000
9 780133 013184